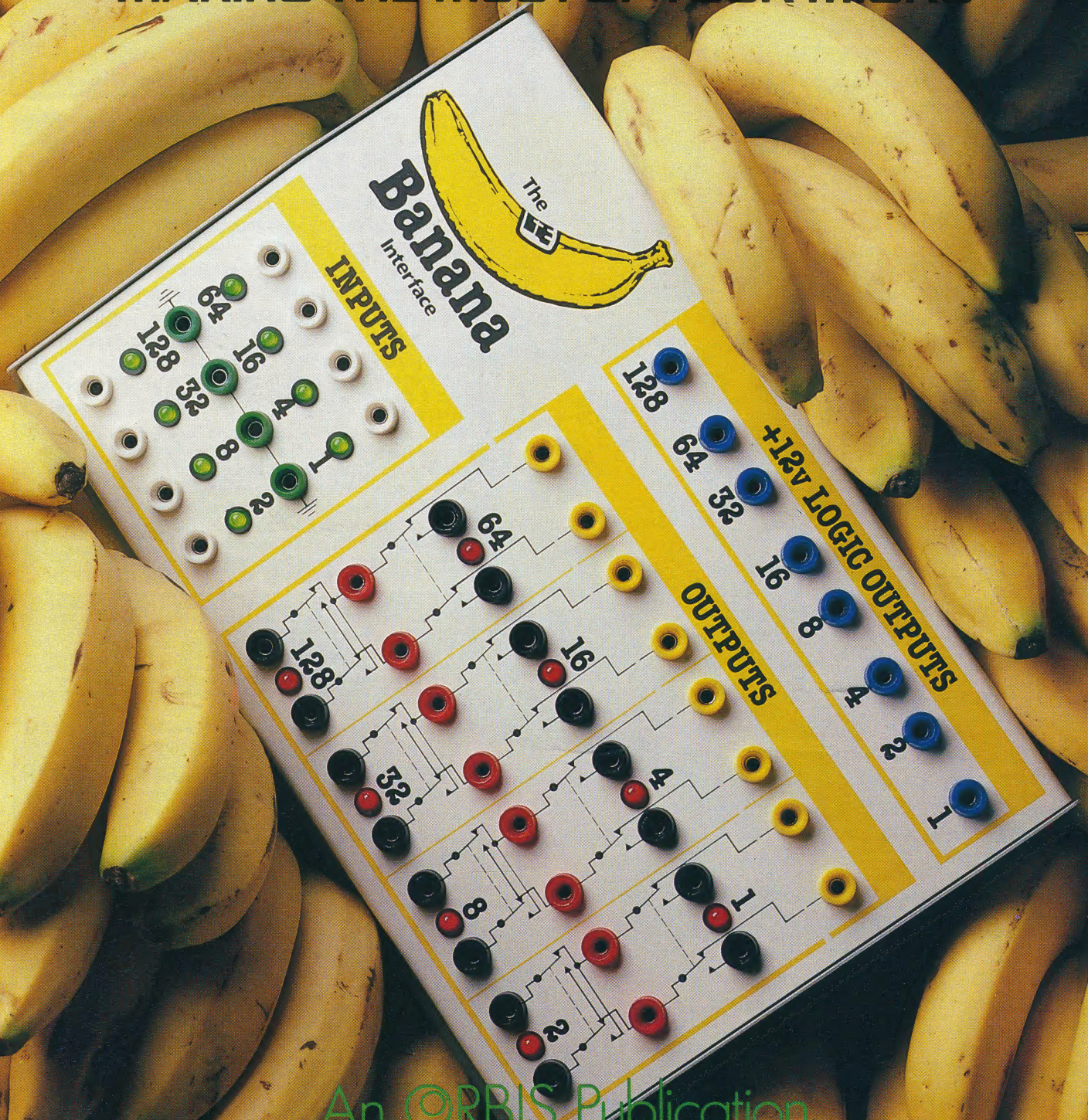
ISSN 0265-2919

90p 65

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

SCOUT AND B-STAR We investigate ways of programming games that have an element of chance, yet also require strategic planning 1281

HARDWARE

BANANARAMA The Banana Interface from Castle Associates allows a wide range of peripherals to be controlled from a computer 1290

SOFTWARE

TRANSIENT COMMANDS CP/M has a range of commands that are loaded from disk when required 1284

NICE LITTLE EARNER Minder has long been a popular television programme. Now a game from DK'tronics puts you in Arthur's hot seat at the Winchester Bar 1300

COMPUTER SCIENCE

DECLARING AN INTEREST We look at PROLOG's use of declarative statements and 'backtracking' 1292

JARGON

FROM PERIPHERAL TO PERSONAL COMPUTER A weekly glossary of computing terms 1289

PROGRAMMING PROJECTS

PROFIT OR LOSS The design of the final algorithm for our simulation game includes the preparations for the homeward journey 1294

MACHINE CODE

SPECTRUM ANALYSIS Our introduction to a series on the Sinclair Spectrum's operating system discusses the best places to put your machine code programs in memory 1296

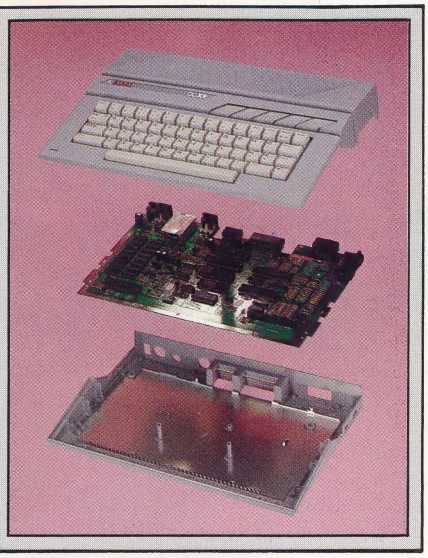
WORKSHOP

MEMORY TRACE We provide a calibration program for the digital tracer we have built over the past two weeks, and give the software to control the device 1286

REFERENCE CARD Another section of the Commodore 64's memory map INSIDE BACK COVER

Next Week

- **In the same price-range as the Commodore 64, the Atari 130XE has 128 Kbytes of RAM. We look at the computer with the enormous memory.**
- **Much of the recent fusion of computers and music is based around the MIDI interface. In our Workshop series we begin to build our own MIDI.**
- **The first part of the complete listing of our New World simulation game will be given.**



QUIZ

1) Suggest two problems you might encounter when placing a machine code program in the Spectrum's printer buffer.
2) Why would a PROLOG program 'backtrack'?
3) What does PIP stand for and what does it do?
4) What is the function of a PIA?

Answers To Last Week's Quiz

1) In order to be able to run CP/M a computer needs an Intel 8080, or equivalent processor, a floppy disk drive and at least 48 Kbytes or RAM.

2) Because the Brother HR-5 is fitted with a Centronics interface, the Spectrum will require an interface to be fitted to the expansion bus to convert Centronics signals.

3) The 'template' method of pattern recognition is limited because the computer can recognise only the shapes that correspond to the template. Futhermore, storing these templates requires a great deal of memory space.

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Deputy Editor** Steve Colwill; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Steve Malone; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Graham Storrs, Geoff Bains, Joe Pritchard, Steve Malone, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA –** Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.

ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.

NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CRISPIN THOMAS

SCOUT AND B-STAR

Futile Search
Although some games can be played intelligently by computers employing searching methods to look ahead a number of moves, there are many games in which searching strategies are not effective; either the game has a random element in it, like Backgammon, or the game tree quickly branches on successive moves to produce a vast number of possible move permutations. To create programs that can play these sorts of games, alternative methods to game tree-searching must be sought

As we more closely examine strategic planning in AI, we consider some improvements as well as alternative game-playing strategies required by games of chance. We also include a non-recursive version of the Numbers Game for Commodore 64 and Sinclair Spectrum owners, which illustrates the principles of alpha-beta minimaxing.

The alpha-beta procedure, explained in the previous instalment, is a big improvement on straightforward minimaxing (since it identifies and removes redundant branches from the game tree); it has been at the heart of the most effective computer chess programs for many years. But recently two alternative strategies have been proposed. One is the Scout algorithm of Judea Pearl and the other is Hans Berliner's B-Star (B*) algorithm.

The essence of the Scout method is to have a highly tuned evaluation function that can be used to reject implausible moves without further search. Only the moves that appear promising need be examined in depth.

The B* method scrutinises the moves at the top level of the tree and tries, as rapidly as possible, to do one of two things:

- Prove that the *apparently* best move is in fact the best available.
- Prove that none of the alternative moves is better.

This twin strategy is implemented by a pair of procedures called ProveBest and RefuteRest, which rely on assigning two values to each node in the tree — one an optimistic evaluation, the other pessimistic. The object is to force the search algorithm to concentrate on areas in the game tree where there is uncertainty and where this uncertainty could affect the final decision.

WHEN SEARCH BREAKS DOWN

It would be wrong to assume that tree-search is the only approach to computer game-playing. There are some interesting games in which the search philosophy seems to break down, regardless of the ingenious tricks adopted to streamline the process. These include many of the most popular card games (most notably bridge and poker) and several board games (such as Go and Go-moku).

These games can be played with varying degrees of skill, and the outstanding human exponents are deservedly called intelligent. Yet all attempts to program them within the tree-search

DAVID LAWRENCE

Tree Search Jargon

Game Tree	Tree structure formed by considering possible moves, followed by opponent's possible replies, and so on.
Ply	One level in the game tree.
Look-ahead	The process of making a game tree.
Backed-up Value	The value assigned to a node in the game tree by looking at the values beneath it, working from the bottom.
Minimaxing	Choosing which value to back up the tree by minimising at odd plies (other to move) and maximising at even plies (self to move).
Alpha-Beta Algorithm	A refinement of minimaxing which eliminates portions of the game tree that cannot possibly affect the outcome at the top level.
Branching Factor	The average number of branches or moves at each level of the game tree. The oriental game of Go has a branching factor of over 200.

framework have hit snags. One reason is that the branching factor is simply too great, generating such a large number of possible move combinations that the computer just cannot handle them all at once.

A more profound answer is that tree-search algorithms are only a crude approximation of the way people attack the problem: experts only mentally search their own game trees, internally constructed in special circumstances, and even then not very effectively.

Hans Berliner, who devised the B* method, developed a backgammon program that beat the world champion in a challenge match in 1980. Yet the program does not search at all, at least not in the conventional sense. If you think about how the game of backgammon is played, you will realise that its game tree is probabilistic: the roll of the dice introduces branches that are under the control of either player, making many of the look-ahead procedures previously outlined difficult to implement.

What Berliner's program is very good at (much better than us) is calculating the odds of various dice and counter combinations occurring during the game. It also has a very sophisticated evaluation function. In fact, it actually employs several functions for different phases of the game and varies smoothly between them as the game progresses.

Go is an Oriental game, played by moving stones on an 18 by 18 grid, with the object of surrounding areas of the grid to gain territory, and of surrounding your opponent's stones in order to remove them from the game. It lacks a chance element, but the branching factor is so immense that search-based techniques are futile.

The best Go programs perceive the board in terms of units higher than just single stones (such as 'strings' and 'armies') that are meaningful groupings to the human eye; one reason why Go programming lags behind chess may be that our understanding of human perception is inadequate. Perhaps the Japanese, who revere the game, will regard Go as a suitable project for their Fifth Generation parallel machines. A successful Go program will certainly push the concepts of artifical intelligence to their limits.

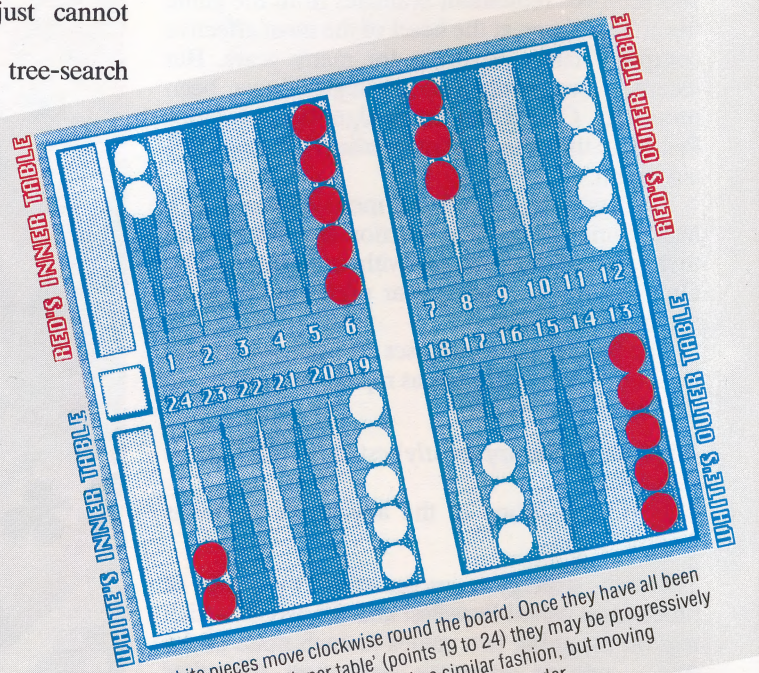
Go programs already exist for the current generation of home computers, and we shall be considering the problems involved in programming the game in a forthcoming game project. These programs, however, do not come close to beating the masters of the game.

Inside The Inner Table

Backgammon is played on a board consisting of 24 triangular 'points' on which the counters are moved, according to the throw of a pair of dice. The object of the game is to move all your pieces round the board and off the end before your opponent.

If the numbers on the dice are different, you can either move one piece the total number of points indicated on the dice, or move two pieces separately according to each individual number thrown. When one piece is moved according to the sum of the dice, it is considered to be making two separate moves and is said to 'touch down' on the boundary point between the two moves.

Points occupied by two or more counters ('men') of one colour may not be moved to or 'touched down' upon by the opposing side. If only one man occupies a point and is 'hit' by an opposing piece, it is sent off the board and must be re-entered by its owner before another move can be made. The illustration shows one move in the game between Hans Berliner's backgammon program and the world champion, Luigi Villa, which took place in Monte Carlo in 1980



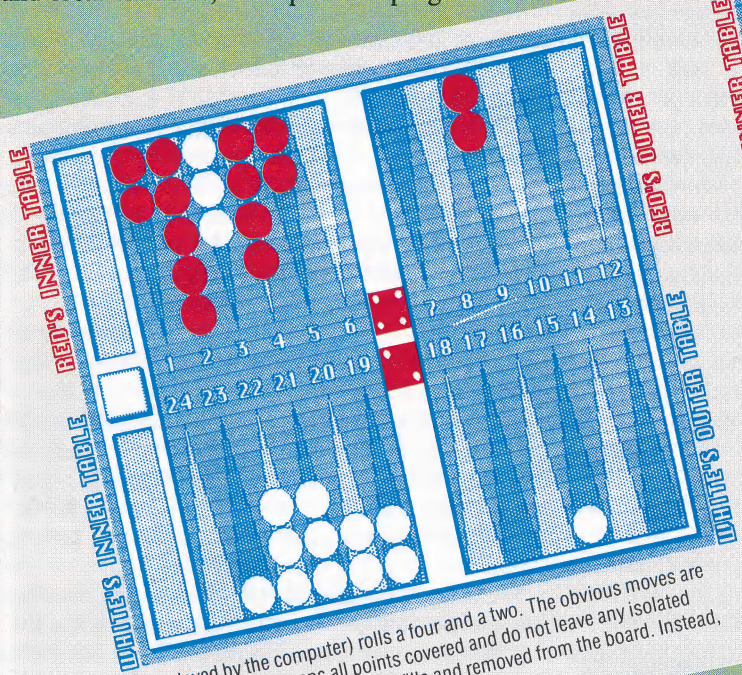
The white pieces move clockwise round the board. Once they have all been entered into white's 'inner table' (points 19 to 24) they may be progressively removed from the board. Red plays in a similar fashion, but moving anticlockwise, following the points in descending order

To illustrate the important concepts of tree searching, we devised an artificial game that was almost pure search. This was given in our previous instalment (see page 1271) and was in the BBC version of BASIC, making heavy use of the recursive functions with parameters.

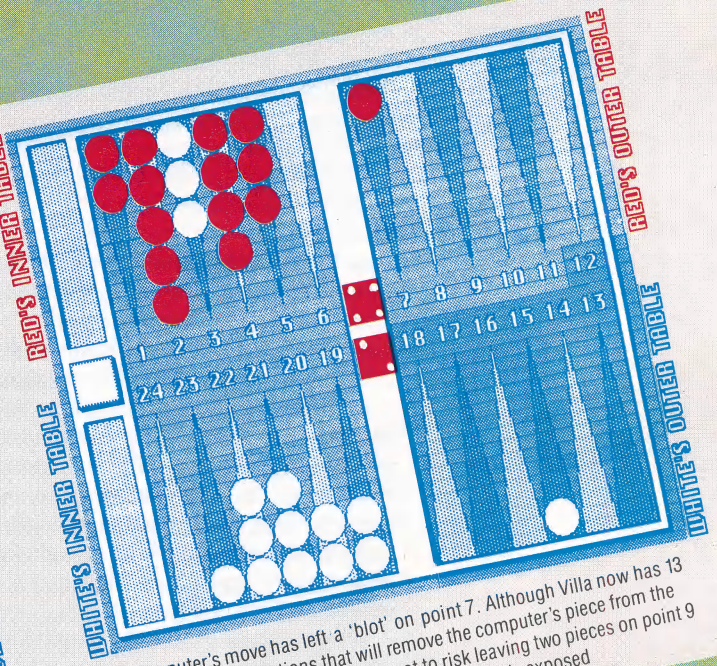
The game allows you and the computer to take turns selecting one of four functions that modify an existing value to produce a new one. You try to reduce the value to -255 while the computer tries to increase it to 255. At each stage the computer uses the alpha-beta minimaxing to select which of the four defined functions will be best for its purposes.

For the benefit of Commodore 64 and Spectrum owners, whose BASICS lack parameters and local variables, we repeat the program in a version that uses only GOSUBs. The parameters and local variables (except D, the depth counter) have been replaced by arrays, DIMensioned on line 1100. D will now act as a kind of stack pointer, keeping track of the array elements currently being accessed.

These two routines differ mainly in that everything has to be subscripted by D, particularly the array A(), used to hold the best value found so far, and B(), used to hold the worst. This ensures that the values of alpha and beta used will apply to the correct levels in the tree.



Red (played by the computer) rolls a four and a two. The obvious moves are 5-1 and 4-2, which keeps all points covered and do not leave any isolated men ('blots') which may be hit by Villa and removed from the board. Instead, the computer decided to move 9-5 and 9-7



The computer's move has left a 'blot' on point 7. Although Villa now has 13 possible dice combinations that will remove the computer's piece from the board, the program decided (wisely) not to risk leaving two pieces on point 9 which, when moved later, could become dangerously exposed

The Numbers Game

```
30 GOSUB 1000:REM INITIALISATION
90 GOSUB 1600:REM INSTRUCTIONS
100 :
110 REM **** MAIN PROGRAM LOOP ****
120 GOSUB 2000:REM PREPARE NEW GAME
130 INPUT"WHO GOES FIRST (1=YOU,2=ME)";H1
140 IF H1<1 OR H1>2 THEN 130
150 REM *** GAME LOOP ****
160 IF H1=1 THEN GOSUB 3000
170 REM ** PERSON'S TURN **
180 GOSUB 3500:REM BOARD DISPLAY
190 H1=1:REM ALWAYS 1 AFTER 1ST CYCLE
200 GOSUB 4000:REM TEST FOR A WIN
210 IF EG=0 THEN GOSUB 5000
220 REM ** COMPUTER'S TURN **
230 GOSUB 3500:REM DISPLAY GAME STATUS
240 GOSUB 4000:REM TEST FOR END OF GAME
250 IF EG=0 AND M<=33 THEN 150:REM LOOP BACK
260 REM **** FINALE ****
270 GOSUB 6000:REM CONGRATULATIONS
280 INPUT"ANOTHER GAME (1=YES,2=NO)";Y
290 IF Y<1 OR Y>2 THEN 280
300 IF Y=1 THEN 110:REM NEW GAME
310 PRINT:PRINT"SO LONG AND THANKS FOR THE GAME"
320 END
330 :
500 REM **** MAXIMISE ****
510 D=D+1:C1=C1+1
520 IF D>=MD OR ABS(V(D))>HI THEN A(D)=V(D):D=D-1:RETURN
530 REM ** ELSE GO DEEPER **
540 P(D)=0
550 REM ** THROUGH TREE **
560 P(D)=P(D)+1:H=P(D):V=V(D):GOSUB 5500:REM MAKE MOVE
570 IF D=1 THEN PRINTCHR$(64+H);" = ";
580 D1=D+1
590 A(D1)=A(D):B(D1)=B(D):V(D1)=V:GOSUB 700:REM CALL MINIMISE
600 IF B(D+1)>A(D) THEN A(D)=B(D+1):K(D)=P(D)
610 IF D=1 THEN PRINTB(D+1);" ";
620 IF P(D)<=3 AND A(D)<B(D) THEN 550
630 IF D=1 THEN BV=A(D):HH=K(D):REM KEEP BEST SO FAR
640 D=D-1:RETURN
650 :
700 REM **** MINIMISE ****
710 D=D+1:C2=C2+1
720 IF D>=MD OR ABS(V(D))>HI THEN B(D)=V(D):D=D-1:RETURN
730 P(D)=0
740 REM ** THROUGH TREE **
750 P(D)=P(D)+1:H=P(D):V=V(D):GOSUB 5500:REM MAKE MOVE
760 D1=D+1:A(D1)=A(D):B(D1)=B(D):V(D1)=V
770 GOSUB 500:REM CALL MAXIMISE
780 IF A(D+1)<B(D) THEN B(D)=A(D+1)
790 IF P(D)<=3 AND B(D)>A(D) THEN 740
800 D=D-1:RETURN
810 :
1000 REM **** INITIALISE ****
1010 BL$=""
1020 REM ** THE FOUR FUNCTION DEFINITIONS **
1030 DEF FNA(X)=2*X-7
1040 DEF FNB(X)=INT(X/2)+1
1050 DEF FNC(X)=4*X+17
1060 DEF FND(X)=3*X-4
1070 LO=-255:HI=255
1080 REM ** ARRAYS USED MY MINIMAX **
1090 D=16
1100 DIM V(D),A(D),B(D),P(D),K(D)
1110 RETURN
1120 :
1600 REM **** INSTRUCTIONS **
1610 PRINT"WELCOME TO THE NUMBERS GAME"
1620 PRINT"I TRY TO MAXIMISE, YOUR JOB"
1630 PRINT"IS TO MINIMISE"
1640 PRINT"TO SEE THE EFFECT OF A MOVE TYPE:"
1650 PRINT"A, B, C OR D. TYPE X TO MAKE IT"
1660 PRINT:RETURN
1670 :
2000 REM **** PREPARATION ****
2010 M=0:V=INT(RND(1)*15)-8:REM INITIAL STATE
2020 EG=0
2030 PRINT"INITIAL STATE = ";V
2040 RETURN
2050 :
3000 REM **** PERSON'S MOVE ****
3010 M=M+1:PRINT
3020 PRINT"YOUR MOVE IS ";
3030 INPUT H$
3040 IF H$="A" THEN PRINT FNA(V):H=1
3050 IF H$="B" THEN PRINT FNB(V):H=2
3060 IF H$="C" THEN PRINT FNC(V):H=3
3070 IF H$="D" THEN PRINT FND(V):H=4
3080 IF H$<>"X" THEN 3020:REM MOVE NOT YET SELECTED
3090 GOSUB 5500:REM MAKE MOVE
3100 RETURN
3110 :
3500 REM **** BOARD DISPLAY ****
3510 PRINT:PRINT"          MOVE";M;" -->";
3520 IF M<1 THEN RETURN
3530 PRINT CHR$(64+H);
3540 PRINT" = ";V:PRINT:RETURN
3550 :
4000 REM **** WIN TEST ****
4010 IF M<1 THEN RETURN
4020 EG=0
4030 IF V<LO THEN EG=-1
4040 IF V>HI THEN EG=1
4050 RETURN
4060 :
5000 REM **** COMPUTER'S MOVE ****
5010 W=V:REM SAVE CURRENT STATUS
5020 M=M+1
5030 MD=6:REM MAX DEPTH
5040 IF M<4 THEN MD=4
5050 IF M>8 THEN MD=8
5060 GOSUB 5200:REM ---> H
5070 V=W:REM RESTORE STATUS
5080 GOSUB 5500 :REM MAKE MOVE
5090 RETURN
5100 :
5200 REM **** MOVE SELECTION ****
5210 BV=LO:D=0
5220 V(1)=V:A(1)=LO:B(1)=HI
5230 GOSUB 500:REM MAXIMISE
5240 H=HH
5250 PRINT:INPUT"PRESS RETURN TO CONTINUE";Q
5260 RETURN
5270 :
5500 REM **** MAKE A MOVE ****
5510 IF H=1 THEN V=FNA(V):RETURN
5520 IF H=2 THEN V=FNB(V):RETURN
5530 IF H=3 THEN V=FNC(V):RETURN
5540 IF H=4 THEN V=FND(V):RETURN
5550 :
6000 REM **** CONGTRATULATIONS ****
6010 PRINT:PRINT"GAME OVER"
6020 IF EG>0 THEN PRINT"I WON IT"
6030 IF EG<0 THEN PRINT"YOU WON IT"
6040 IF EG=0 THEN PRINT"GAME DRAWN"
6050 RETURN
```

TRANSIENT COMMANDS

As we continue our series on the CP/M operating system, we discuss the 'transient' commands, which provide us with much manipulative power when managing files and using peripherals. We illustrate these by progressing step by step from the 'initialisation' of the computer, through to the specific functions of several commands.

In the initial instalment of our series on CP/M, we looked at the development of the system and how it was structured. We saw that while a number of CP/M commands are kept permanently in RAM, others — which are held on disk — are LOADed into the system only when needed and then discarded. These 'transient' commands allow the user to communicate with the disk drives, manage files and control other peripheral devices.

On power-up the computer will run through a series of checks to ensure that all the components of the system are present and correct. The processor will also open channels to any peripherals that may be attached. The computer will send codes to the various peripherals partly to check whether they are working and partly to ready them to receive data. This whole process is known as 'initialisation'.

When initialising some peripheral interfaces, the computer will wait for a message to return from the peripheral before it proceeds with any further action, such as activating the BASIC ROM. An example of this can be seen when a games cartridge is fitted, which in effect takes over the computer's operating system and performs all the input and output to the processor itself, without referring to the BASIC ROM. Similarly, when a CP/M-compatible computer opens a channel to its disk drive, it will wait until a message returns from the drive.

Extension	Comment	Example
ASM	Required for assembly language source file	CODEPROG.ASM
BAK	Duplicate text file created by the editor	MEMO.BAK
BAS	Appended to a BASIC source file	PROG.BAS
COM	Necessary extension for a command or transient file	PIP.COM
HEX	Assigned to a machine level hexadecimal file	GRAPHIC.HEX
INT	Given to a compiled BASIC program	GAMES.INT
PRN	Needed to produce assembly language program listings	CODEPROG.PRN
SUB	File used to execute batch commands	SPOOLER.SUB
$$$	Temporary file created by the editor	PHONE.$$$

The result of the computer's initialisation on the disk drive is that the read/write head of the device is forced to read the first track on the CP/M system disk. If no disk is present, the drive will continue turning until a disk is inserted. Track zero contains the 'bootstrap loader' program, which in turn gives instructions to the computer informing it to begin loading the rest of the CP/M program into memory. If no bootstrap program is present on track zero, the computer will generate a DOS error.

After CP/M has finished being LOADed into the computer, a flashing cursor will appear on the screen next to a symbol A> (although some machines use 0A>). This is known as a 'prompt' and indicates that the system is ready to receive a command. It also informs you which disk drive CP/M is currently using and where it expects to read and write information. The drive that CP/M is accessing at present is known as the 'currently logged drive'. In this case, since we have only just LOADed CP/M, we are still in drive A.

This is fine if we have only one disk drive or wish to use only one. However, if we have two disk drives and wish to use both of them we have to tell CP/M to look at drive B. This is performed by the command B: and pressing RETURN. CP/M will then check to see whether drive B is present and whether a disk is actually in the drive. If so, the prompt B> will appear on the screen, indicating that we are now in drive B. But we are not limited to two drives. Up to four can be accessed by CP/M, the others being referred to as C and D respectively.

Although it is unusual for four separate disk drives to be fitted to a single machine, many computers, such as the 380Z and 480Z from Research Machines, have double-sided drives. In such cases, it is normal for the top sides of the drives to be called A and B and the bottom sides C and D.

After logging on to CP/M, it is usual to ascertain what files are available. To do this we have to examine the directory of the disk by entering the command DIR or dir — CP/M does not distinguish between upper and lower case letters. This will display a list of files, including all the commands that can be LOADed and RUN from CP/M.

It might appear that having to LOAD and RUN a 'command file' simply in order to, for example, return status information about a file held on disk is an unnecessarily long-winded process. Why shouldn't commands be loaded into RAM on power-up, to be directly executed when needed? The main reason for this method, apart from the obvious one of saving memory space, is to facilitate compatibility between different systems.

As you might imagine, CP/M contains a number of commands that enable us to manipulate files held on disk. Although CP/M applications have to be standard in order to be portable from one machine to the next, many manufacturers over the years have adapted the system to suit the needs of their own computers. For example, there is no standard disk formatting command in CP/M, so each manufacturer adds this particular function to the list of CP/M transient commands, resulting in many systems working differently.

As we saw in the previous instalment, a CP/M file consists of two parts. The file name consists of a primary name, a full stop and an extension. The extension affects

the way a file is LOADed into the computer. On the CP/M systems disk are a number of command files containing the CP/M transient commands, all having the extension .COM. This means they will be run automatically when loaded into the computer.

THE STAT COMMAND

In order to LOAD a command file, you simply type in the primary name, press RETURN and the command program will execute. Other files with different extensions are loaded differently and we shall be looking at these later on in the course.

We have previously examined the use of the DIRectory command. There is a related transient command in CP/M known as STAT, which provides you with additional information about the disk and the files held on it. When it is executed, STAT will display the amount of free memory available for new files. The command will also display certain information about the disk itself, such as how much memory is still available for other files and what sort of read/write facilities are permitted. For example, the message R/W means that the disk can be read and written to, while R/O means that the disk can only be read or, rather, that it is write protected.

Furthermore, the STAT command can also 'lock' a disk so that it can only be read. This is achieved by typing STAT D:=R/O, where D refers to the Drive character. Any subsequent attempt to write to this particular disk will result in a BDOS error. The command will also display the filename extensions and the number of logical file sectors used by each record, and it is possible by using STAT to examine the size of individual files simply by typing STAT followed by the filename. STAT can also be used to examine and, if required, change the status of the peripherals that may be attached. As an example, the command STAT DEV will display a list of all the input/output devices (including the screen) that are connected.

One of the most important features of a disk operating system is the ability to transfer files from one disk to another. In CP/M, this operation is performed by PIP (Peripheral Interchange Program). As its name suggests, this command does more than copy from disk to disk; it also allows files to be output to the printer or other input/output devices.

In order to use PIP, the command must be loaded and run by typing in PIP and pressing RETURN. Notice that the prompt now changes to an asterisk, indicating that PIP is running and awaiting the next command. The format for using PIP is: D:COPYNAME = D:SOURCENAME. As an example, let us assume that we have a textfile HCAC.TXT on one disk and wish to copy it on to a second disk under the name WORK.TXT. We must remove the systems disk from the drive and place the disk containing HCAC.TXT in drive A, and place the second disk in drive B. So to copy HCAC.TXT from drive A to drive B, the format is: B:WORK.TXT = A:HCAC.TXT.

Note that when using PIP, the target drive is placed first in the command before the source drive that follows, and that when referring to non-command files, the entire name, including the extension, must be included. Once the operation has been completed, you can check whether the file has been successfully copied by examining the directory on drive B.

You can still use PIP if you have only one disk drive but the source and destination drives will, of course, both be A. CP/M will copy sectors of the file into memory and inform you when the disks should be changed so that the sectors can be dumped onto the destination disk.

Should we wish to send HCAC.TXT to the printer, a similar form of PIP is used. In this case, we issue the command: PIP LPT:=B:HCAC.TXT. There are several things to notice about this command, the first of which is that PIP was not loaded separately from the rest of the command. This is quite possible in CP/M since you do not necessarily have to wait for PIP to be loaded before anything else can take place.

Secondly, the destination device is not a drive channel but a printer, which is why we use the command LPT ('Line PrinTer'). In this command, we have also included the drive letter indicating where the file HCAC.TXT can be found. If drive B is currently logged, this inclusion will not be necessary. In fact, under some versions of CP/M, it is not needed even if we are in drive A. This is because CP/M will search all available disk drives before generating a 'file not found' error.

In transferring the file from one disk to another, we also changed the name of the file. This is not a requirement of PIP; a file can be copied across and still retain the same name. However, supposing we wanted to change the name of a file on a disk without having to copy it. In this case, we would use the RENname command, so if we wanted to change the name of WORK.TXT to HCAC.TXT, we would use the command REN HCAC.TXT=WORK.TXT, remembering that the target name is always placed first. This will create a new file name in the directory and delete the previous one.

Similarly, when we wish to erase a file on a disk, ERA followed by the filename is typed in. As with all such destructive commands, care must be taken with its usage. CP/M must know which disk you are referring to, otherwise you might end up deleting the wrong version of the file. So, although it is not necessary, it is nevertheless good practice always to include the name of the drive in the command; for example, ERA B:HCAC.TXT.

In the next instalment of this series we shall examine some more CP/M commands and take a look at some of the control characters that are available to the system. We shall also highlight the uses of file extensions.

ASM — The CP/M Assembler

CP/M, as implemented on most machines, includes its own assembler utility called ASM. Although frequently ignored by programmers it is a powerful assembler with provision for conditional assembly, as well as all the standard assembler directives. The only real drawback in using ASM is that it was originally developed for use on the 8080 assembler, which was later replaced by the Z80. Although object code generated for the 8080 will run on the Z80, you will find that standard Z80 mnemonics will not always be accepted by an 8080 assembler such as ASM. You will find a full list of deviations in Rodnay Zaks' book 'Programming the Z80', published by Sybex. If, however, you intend to make regular use of ASM on a Z80 machine we advise you to contact Digital Reseach, who can provide a set of macro-routines to run with ASM that will enable you to generate native Z80 code using familiar mnemonics

MEMORY TRACE

Having completed the construction of the digital tracer, we can turn our attention to its calibration and the development of software for it. We give a program which allows us to use the tracer to generate screen displays on the BBC Micro that can be saved to or loaded from disk and tape.

The first step in developing software for the tracer is to calibrate the hardware. Running the Calibration program given here provides us with four digital potentiometer values that correspond to four critical arm positions: arm 1's zero position, arm 2's zero position, arms 1's 90° position and arm 2's 90° position. These values should be entered into the digital tracer program at lines 1200, 1210, 1240 and 1250, as part of the define_parameters procedure. Two other parameters that may vary from one tracer to another are the lengths of the arms, and these should be entered into the program at lines 1220 and 1260. Note that the arm lengths should be in millimetres and that arm 2 includes the distance from the end of the arm to the cross-hairs.

READINGS TO ANGLES

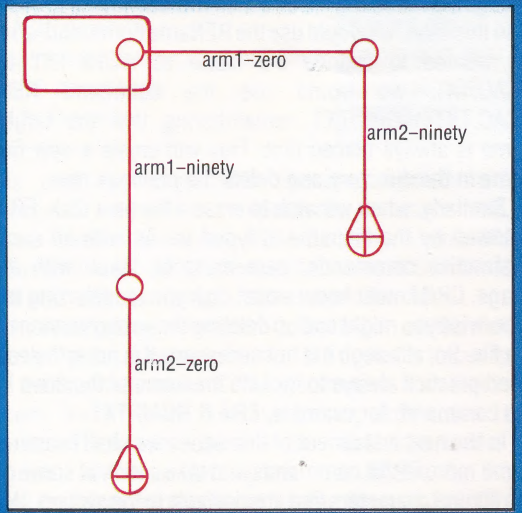
Having obtained these calibration values, we can use them to convert the digital readings provided by the potentiometers (via the BBC Micro's on-board analogue-to-digital converter) into angles. This angular information can then be converted into horizontal and vertical displacements of the sight from the arm's mounting point. In general, an angle can be calculated using the following formula:

Calibration

Set the tracer board up so that the plastic box is in the upper left corner. In order to calibrate the tracer successfully, we need four pieces of information: Arm 1's 0° position, Arm 1's 90° position, Arm 2's 0° position and Arm 2's 90° position. With the calibration program running, move the tracer arm so that Arm 1 is parallel with the top edge of the board and Arm 2 is perpendicular.

Note the digital values for Arm 1 and Arm 2 on a piece of paper. Move the arm assembly so that Arms 1 and 2 are now in line and parallel to the left-hand edge of the board. Again note the readings for Arms 1 and 2. Press a key on the keyboard to move into angle mode.

Calibration Positions

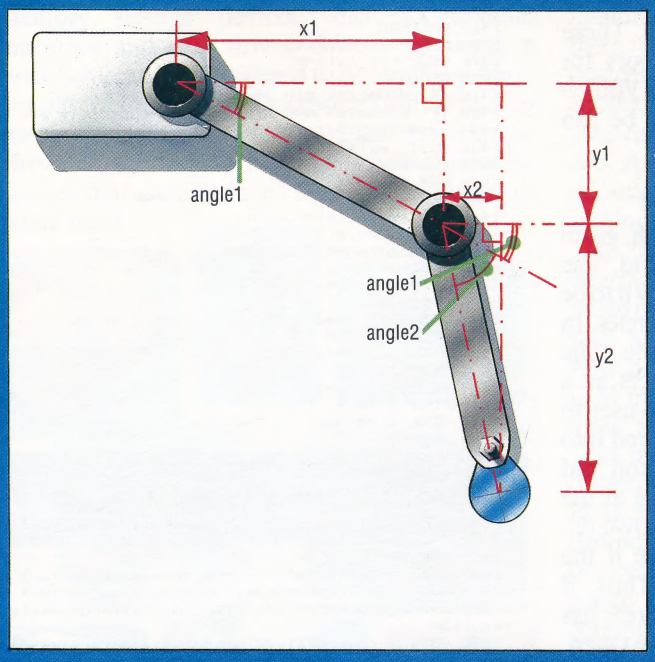


Enter the four readings you have noted in response to the prompts and then check that the angles displayed for both arms are correct. If not, repeat this process until these angles are correct. Keep a note of the four digital arm values because they will be needed for the tracer program that we will be providing in the next instalment.

```
1000 REM **** BBC TRACER CALIBRATION ****
1010 PROCcalibrate
1020 PROCdisplay_angle
1030 END
1040 DEF PROCcalibrate
1050 CLS
1060 PRINT TAB(5,10);"ANGLE 1";TAB(20);"ANGLE 2"
1070 REPEAT
1080 PRINT TAB(5,12);SPC(40)
1090 PRINT TAB(5,12);ADVAL(1);TAB(20)ADVAL(2)
1100 PROCdelay(600)
1110 A$=INKEY$(1)
1120 UNTIL A$<>""
1130 ENDPROC
1150 DEF PROCdisplay_angle
1160 PROCdefine_angles
1170 CLS
1180 PRINT TAB(5,10);"ANGLE 1";TAB(20);"ANGLE 2"
1190 REPEAT
1200 PROCcalc_angles
1210 PRINT TAB(5,12);SPC(40)
1220 PRINT TAB(5,12);angle1;TAB(20)angle2
1230 PROCdelay(1000)
1240 A$=INKEY$(1)
1250 UNTIL A$<>""
1260 ENDPROC
1280 DEF PROCdefine_angles
1290 CLS
1300 PRINT TAB(5,10);:INPUT"1st arm zero position"
;one_zero
1310 PRINT TAB(5);:INPUT"1st arm ninety position"
;one_ninety
1320 PRINT TAB(5);:INPUT"2nd arm zero position"
;two_zero
1330 PRINT TAB(5);:INPUT"2nd arm ninety position"
;two_ninety
1340 one_factor=90/(one_ninety-one_zero)
1350 two_factor=90/(two_ninety-two_zero)
1360 ENDPROC
1380 DEF PROCcalc_angles
1390 angle1=INT((ADVAL(1)-one_zero)*one_factor)
1400 angle2=INT((ADVAL(2)-two_zero)*two_factor)
1420 DEF PROCdelay(delay)
1430 LOCAL I
1440 FOR I=1 TO delay:NEXT I
1450 ENDPROC
1460 PROCdelay(1000)
```

KEVIN JONES

Arm Geometry



The horizontal and vertical displacements of the tracer's sight can be calculated from the angles provided by the two potentiometer readings, using simple geometric calculations based on the two right-angled triangles shown. The lengths of x1 and y1 correspond to the length of arm 1 multiplied by COS(angle 1) and SIN(angle 1) respectively. As the angle provided by the second potentiometer is relative to arm 1's position, the lengths of x2 and y2 are a little more difficult to calculate. The required angle in the second triangle is not angle 2, but angle 1 + angle 2 — the COS and SIN functions being used together with the length of arm 2 to calculate the horizontal and vertical displacements. Overall displacement of the sight from the arm pivot can be found by adding x1 and x2, and y1 and y2

KEVIN JONES

```
angle=(ADVAL(n)-armzero)*90/(armninety-
  armzero)
```

As the second part of this expression is a constant, it is best to calculate the value at the beginning of the program and use this in all subsequent angle calculations. If we calculate:

```
factor=90*/(armninety-armzero)
```

we can then re-express the formula as:

```
angle=(ADVAL(n)-armzero)*factor
```

The procedure calc_xy calculates the current angles taken up by the two potentiometers and uses these to calculate the x and y displacements of the sight. The geometry involved is fairly straightforward. With reference to the arm geometry diagram, x1 and y1 are related to the length of arm 1 by the SIN and COS functions:

```
x1 = COS(angle1)*length_arm1
y1 = SIN(angle1)*length_arm1
```

Calculation of x2 and y2 is a little more tricky, as angle2 is at zero when arm 2 is in line with arm 1. The angle in the corner of the right-angled triangle is therefore angle1 + angle2 and:

```
x2 = COS(angle1+angle2)*length_arm2
y2 = SIN(angle1+angle2)*length_arm2
```

The total x and y displacements can be found by adding x1 and x2, and y1 and y2, respectively.

The formulae used within the procedure calc_xy are slightly different from those given above. The arm length variables are replaced by scaling variables. These variables are related to the two arm lengths, but also incorporate a factor so that the x and y displacements are scaled to fit in with the BBC Micro's graphics co-ordinate system.

SAVING AND LOADING SCREENS

The most obvious way to save displays created on the screen is to perform a machine code save of the RAM area used by MODE 1. As long as the screen has not been scrolled since the last CLS then this area will be from &3000 to &7FFF. However, there is a drawback to using the *SAVE command in that file names cannot be passed to the command using a string variable. For example, the command:

```
*SAVE file$ 3000 8000
```

will save the file with the name file$ rather than the string name held in the variable file$. This means that the user cannot easily save screen displays on disk or tape under different filenames. Fortunately, there is a way around this problem, using the OSCLI operating system call. This command can be used to execute a block of ASCII codes held in memory, as though the corresponding characters had been typed in directly from the keyboard. To use this call we must pass the start address of the ASCII block in lo-byte/hi-byte form to the X and Y registers. To perform *SAVE and *LOAD, we therefore assemble the command (complete with the filename provided by the user) in a string and POKE the ASCII values of the characters making up the string into a reserved block of memory. After setting the X and Y registers to point to the start of this block, the command stored there is performed by calling OSCLI at address &FFF7. This is the function of the procedure oscli_command at line 2170.

In addition to saving and loading screens, the screen can be cleared by typing C and a return to the main menu made by typing M. The second selection from this menu cannot be successfully

made at this stage, as this will form the second part of the program and will be given in the next instalment. New foreground colours can be selected at any time by pressing 1 2 or 3. These colours are the default foreground colours for MODE 1, and can be changed using the VDU 19 command. The program given here will be run correctly in freehand mode.

DIGITAL TRACER PROGRAM

The first half of the Digital Tracer program, given here, allows the tracer to be used freehand. The second half includes routines that will allow it to be used to produce single points, lines and circles. In freehand mode, six features are available. The push-button, mounted on the tracer, acts as a 'pen-up/pen-down' control, allowing the user to toggle between the two. The button is wired into the analogue port like a joystick fire button and can be detected from software by looking at the lowest two bits of the value returned in ADVAL(0). Each of the two bits will be set to one if the corresponding fire button is pressed. Thus, if (ADVAL(0)AND3)<>0 then a fire button press has occurred and the appropriate action can be taken. In this case a variable called togflag will toggle between 0 and 1 on each successive button press. The value of togflag is tested in the procedure draw and a decision made either to DRAW or MOVE to a new position is made (depending on the value).

A small cross is used as a cursor to show the current position of the tracer sight on the screen. This can be rubbed out and moved without disturbing the background data by using the Exclusive-OR plotting mode. Any lines drawn in this mode (selected by GCOL3) can be erased by drawing the line again in exactly the same position. Thus, the procedure draw calculates the new cursor co-ordinates and then calls a cursor drawing procedure to rub out the old cursor by drawing over it in Exclusive-OR mode. The graphics cursor (invisible to us) is then MOVEd to the old cursor position and a line is drawn or a move is made to the new co-ordinates, depending on the value of togflag. Finally, the current co-ordinate values are stored in oldx and oldy in preparation for the next time the draw procedure is called.

```
    6 REM ** BBC DIGITAL TRACER **
 1080 PROCdefine_parameters
 1090 REPEAT
 1100 MODE 1
 1110 REM CURSOR OFF
 1120 VDU 23,1,0;0;0;0;
 1130 PROCmenu
 1140 CLS
 1150 IF ans$="1" THEN PROCfreehand ELSE PROCelastic
 1160 UNTIL endflag=1
 1170 END
```

The program provides initial data, including tracer arm lengths and potentiometer calibration readings

```
 1190 DEF PROCdefine_parameters
 1200 one_zero=14820
 1210 one_ninety=45020
 1220 one_length=250
 1230 one_factor=90/(one_ninety-one_zero)
 1240 two_zero=450
 1250 two_ninety=25300
 1260 two_length=222+28
 1270 two_factor=90/(two_ninety-two_zero)
 1280 REM ** MM TO GRAPHICS COORDS CONVERSION FACT
ORS **
 1290 scale=1023/460
 1300 one_scale=scale*one_length
 1310 two_scale=scale*two_length
 1320 REM SCAN 2 ADC CHANNELS ONLY
 1330 *FX 16,2
 1340 endflag=0:togflag=0:colour=1
 1350 DIM save% 30:DIM x(3),y(3)
 1360 ENDPROC
 1380 DEF PROCfreehand
 1390 PROCfree_inform
 1400 REPEAT
 1410 IF (ADVAL(0) DIV 256)<>0 THEN PROCdraw
 1420 IF (ADVAL(0) AND 3)<>0 THEN PROCtoggle_pen
 1430 ans$=INKEY$(1):IF ans$<>"" THEN PROCfreepress
 1440 UNTIL exitflag=1
 1450 ENDPROC
 1470 DEF PROCdraw
 1480 PROCcalc_xy
 1490 PROCcursor(oldx,oldy)
 1500 MOVE oldx,oldy
 1510 IF togflag=0 THEN DRAW x,y ELSE MOVE x,y
 1520 PROCcursor(x,y)
 1530 oldx=x:oldy=y
 1540 ENDPROC
 1560 DEF PROCcursor(cx,cy)
 1570 GCOL 3,3:REM EOR PLOT MODE
 1580 MOVE cx,cy-16
 1590 PLOT 1,0,32
 1600 PLOT 0,-16,-16
 1610 PLOT 1,32,0
 1620 GCOL 0,colour
 1630 ENDPROC
```

The following procedure performs the necessary calculations to convert the potentiometer readings into horizontal and vertical components x and y

```
 1650 DEF PROCcalc_xy
 1660 angle1=RAD((ADVAL(1)-one_zero)*one_factor)
 1670 angle2=RAD((ADVAL(2)-two_zero)*two_factor)
 1680 x=one_scale*COS(angle1)+two_scale*COS(angle1
+angle2)
 1690 y=1023-(one_scale*SIN(angle1)+two_scale*SIN(
angle1+angle2))
 1700 ENDPROC
 1720 DEF PROCmenu
 1730 exitflag=0:endflag=0
 1740 PRINT TAB(5,10);"Please choose"
 1750 PRINT TAB(5);"1...Freehand mode"
 1760 PRINT TAB(5);"2...Elastic mode"
 1770 PRINT:PRINT TAB(5);"Press 1 or 2"
 1780 REPEAT:ans$=GET$:UNTIL ans$="1" OR ans$="2"
 1790 ENDPROC
 1810 DEF PROCtoggle_pen
 1820 togflag=1-togflag
 1830 REPEAT UNTIL (ADVAL(0) AND 3)=0
 1840 ENDPROC
 1860 DEF PROCfreepress
 1870 IF ans$="C" THEN CLS:PROCfree_inform:ENDPROC
 1880 IF ans$="M" THEN exitflag=1
 1890 IF ans$="S" THEN PROCsave_screen:PROCfree_in
form:ENDPROC
 1900 IF ans$="L" THEN PROCload_screen:PROCfree_in
form:ENDPROC
 1910 PROCcolour_change
 1920 ENDPROC
 1940 DEF PROCfree_inform
 1950 PROCcalc_xy:oldx=x:oldy=y:PROCcursor(oldx,oldy)
 1960 PRINT TAB(1,1);SPC(79)
 1970 PRINT TAB(1,1);"S=Save  L=Load  M=Menu  C=Clear"
 1980 GCOL 0,1:MOVE 0,920:DRAW 1280,920
 1990 ENDPROC
 2010 DEF PROCcolour_change
 2020 ans=VAL(ans$)
 2030 IF ans<1 OR ans>3 THEN ENDPROC
 2040 colour=ans
 2050 ENDPROC
```

These procedures enable the screen area to be saved or loaded as a RAM area using the *SAVE and *LOAD commands indirectly via an OSCLI call

```
 2070 DEF PROCsave_screen
 2080 PROCcursor(x,y):REM CURSOR OFF
 2090 REPEAT
 2100 PRINT TAB(1,1);SPC(79)
 2110 INPUT TAB(1,1);"SAVE FILENAME";file$:file$=f
ile$+".S"
 2120 UNTIL LEN(file$)<8
 2130 file$="*SAVE "+file$+" 3000 8000"
 2140 PROCoscli_command(file$)
 2150 ENDPROC
 2170 DEF PROCoscli_command(a$)
 2180 FOR I=0 TO LEN(a$)-1
 2190 save%?I=ASC(MID$(a$,I+1,1))
 2200 NEXT I
 2210 save%?I=13:REM ADD CR
 2220 X%=save% MOD 256:Y%=save% DIV 256
 2230 CALL &FFF7:REM CALL OSCLI
 2240 ENDPROC
 2260 DEF PROCload_screen
 2270 REPEAT
 2280 PRINT TAB(1,1);SPC(79)
 2290 INPUT TAB(1,1);"LOAD FILENAME";file$:file$=f
ile$+".S"
 2300 UNTIL LEN(file$)<8
 2310 file$="*LOAD "+file$
 2320 PROCoscli_command(file$)
 2330 ENDPROC
```

P

PERIPHERAL

This is usually any device that is attached to a computer and includes all input/output as well as memory storage devices, such as cassette machines and disk drives. Strictly speaking, a *peripheral* can be considered to be anything that is not part of the Central Processing Unit, including on-board devices such as sound and video chips. However, the term is more generally applied to devices that are interfaced to the main computer board itself.

Add-Ons All Around
The number of peripherals available for home computers is growing at a remarkably rapid rate. The vast range now on the market enables users to 'customise' their computers to their own requirements. Peripherals can range from joysticks for games players to highly sophisticated communications modems that can link you up to other computers anywhere in the world

As the microcomputer market now shows every sign of becoming saturated, manufacturers are turning their attention more to the production of peripherals, enabling users to upgrade and customise their present systems. This is in part due to the recognition that the first period in the development of the home computer market — in which machines are purchased either to learn about computers or to play games — is coming to an end. Presently, users require machines for more 'serious' applications such as word processing and database management and so, apart from cassette decks and joysticks, the next purchases for many people are printers and mass storage devices — disk drives and microdrives, for example. It is toward these devices that manufacturers are now directing their energies.

PERIPHERAL INTERFACE ADAPTOR

A *peripheral interface adaptor* (PIA) enables a computer and a peripheral with incompatible interfaces to be connected and operate together. The use of PIAs is widespread throughout the computing industry, the main reason being that manufacturers tend to build unique interfaces into their machines ensuring that only their own devices are compatible. Thus if you wish to attach another manufacturer's piece of equipment to your machine, a PIA has to be fitted between the two. A PIA, therefore, must control each flow of information and convert any codes that may be incompatible.

In the business computer market, peripheral interface adaptors are undoubtedly used mostly to attain compatibility with IBM machines. IBM has such an enormous share of the market that its systems have become industry standards. Non-IBM machines without the ability to conform to these standards will inevitably be less popular. Thus many third party manufacturers make their computers capable of accommodating whatever PIAs may become necessary. At the home computer level, the emphasis is more on providing devices to enable computers, which are often poorly equipped with suitable interfaces, to communicate with standard peripheral devices. Examples of PIAs in this context are the floppy disk drive interfaces for the Spectrum and the parallel printer or cassette interfaces for the Commodore 64.

PERMUTATION

The elements of a set can be arranged in any order, and these arrangements are called *permutations*. For example, a set containing the first three letters of the alphabet has six possible permutations — abc, acb, bac, bca, cba and cab. The number of different permutations possible for a set with n elements is given by n! (pronounced: 'n factorial'). The formula for calculating the factorial of n is:

$$n! = n \times (n-1) \times (n-2) \ldots \times 1$$

As an example, a set with five elements has 5! ($5\times4\times3\times2\times1$) or 120 different permutations.

PERSONAL COMPUTER

A *personal computer* is a micro designed to be operated by a single user and able to carry out a wide variety of tasks. Although it can be configured as part of a network, a personal computer is generally a stand-alone device consisting of a keyboard, monitor, CPU with associated RAM and some form of backing storage device, such as a cassette deck or disk drive. Normally used to refer to business machines (especially the IBM PC), the term can refer to any single-user microcomputer from the Sinclair ZX81 upwards.

Personal computers have only become feasible in the last decade or so with the miniaturisation of computer circuitry and the associated drop in price of computer power. This means the personal computer is now small enough to fit comfortably on a desk-top and its cost can justify the machine being dedicated to a single user.

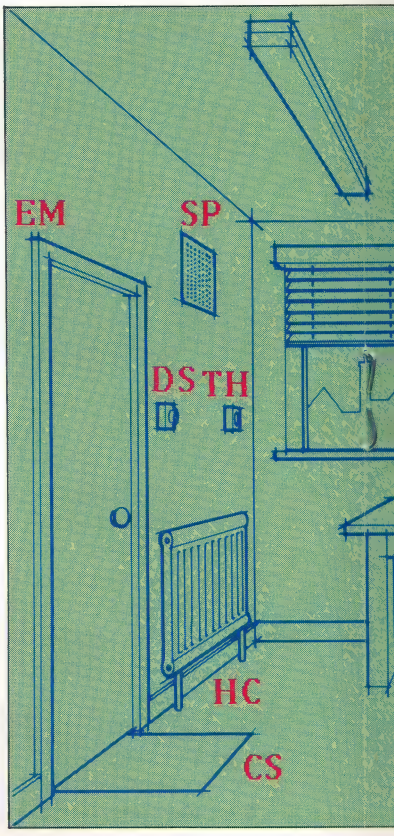
CRISPIN THOMAS

Sensitive Banana
The Banana Interface, so called because of the banana plugs that it uses, enables analogue electrical devices to be interfaced with a computer (which uses digital signals). In this way, feedback systems can be constructed that allow the computer to control — and make responses to — events in the outside world

BANANARAMA

Under Your Skin
The Banana Interface can be used for a wide variety of control applications both at home and in the laboratory. For example, the environment of the room shown here can be placed almost entirely under the control of a computer connected to a Banana Interface. A number of sensing devices, such as thermostats or light sensors, can be fitted into the room. Information received from these devices can be analysed by the computer and compared with preprogrammed optimum ranges. Any adjustments to the environment that need to be made (such as raising the temperature or lowering the blinds and turning on the lights in the evening) can then be performed through the output lines of the Banana

CS Contact Sensor
HC Heating Control
DS Dimmer Switch
EM Electric Motor
SP Alarm Speaker
SD Smoke Detector
TH Thermostat
PC Photosensitive Cell



The Banana Interface, produced by Castle Associates and aimed predominantly at the education market, is a solidly built addition to the field of computer-controlled devices. Though perhaps a little limited for the home micro user, the Banana can teach important concepts by powering such devices as turtles and robot arms.

We have already discussed a number of devices that enable computers to control machinery, as well as illustrated how numerous devices can be controlled in our Workshop series. One of the latest on the market, aimed primarily at schools, is the Banana Interface, designed to enable a number of different projects to be controlled from the BBC Micro or Commodore 64.

Essentially, the concept behind the Banana Interface is identical to that of the buffer box we constructed in the Workshop series (begun on page 523), although the Banana is somewhat more sophisticated. By manipulating the binary bit patterns contained in the data and data-direction registers of the computer, 5v signals can be transmitted to external devices. Furthermore, because the user and printer ports are bi-directional, the computer can receive signals from the peripheral that enable it to 'know' something about the status of the device — its position, for example — or whether any trigger switches have been activated. Thus by a combination of input and output signals, it is possible for the computer to accurately guide a robot and react to the changing circumstances of the robot's environment.

The interface is connected to the computer via a length of ribbon cable that plugs into the printer and user ports. The other end of the cable is fitted onto a 40-way connector on the interface itself. In order to run the Banana Interface, two DC power supplies are required, the first of which is a 5v current supplied from the computer via the ribbon cable. This current is used to run the logic in the interface along the data lines.

The second power source is supplied externally through a pair of minijack sockets on the back of the device next to the ribbon cable interface. Unfortunately, the manufacturer, Castle Associates, has decided not to provide a 12v

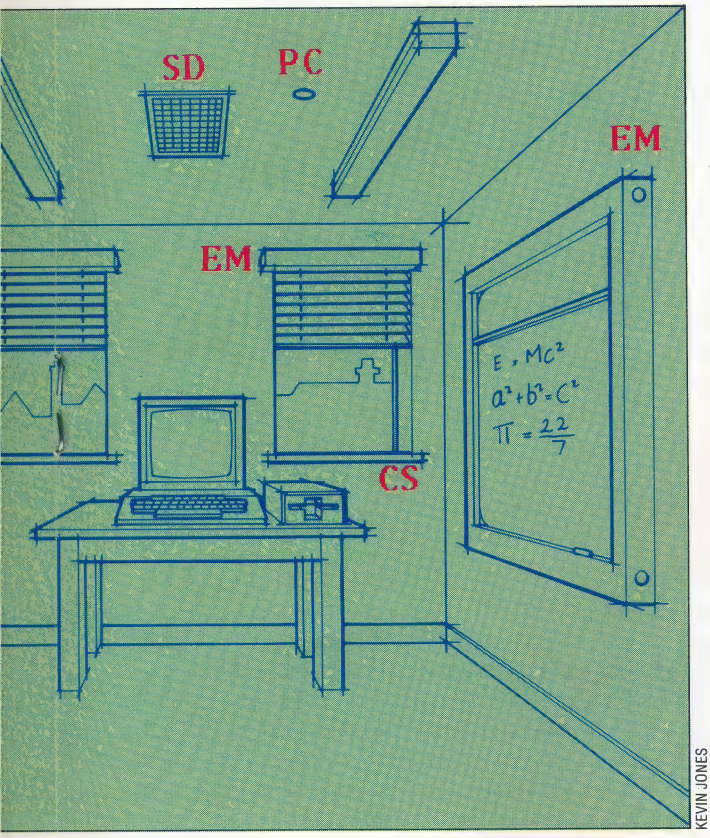
supply with the interface. The reason for this is that the Banana is intended primarily for use in schools, which have ample supplies of the appropriate current in their laboratories and workshops. So, because schools do not require an external power supply, the company has opted to exclude it and keep down the price of the device.

This means that individuals considering purchasing a Banana are faced with the prospect of having to find their own power source. As 12v supplies with small amperages are not readily obtainable, the interface will probably have to be run from a number of batteries connected in a series (the company recommends an amperage of 600mA, though the machine we used ran without difficulty on a 10v one-amp supply).

When the Banana Interface was in the planning stages, Castle Associates consulted Craft, Design and Technology teachers. This had a beneficial effect on the specification of the device, particularly where school use was concerned. With the entire casing made of steel, the interface is one of the most rugged devices available for any home computer.

This concern for durability has also extended to the internal components of the interface. The printed circuit board has been soldered to the jack sockets, holding the delicate board firmly in place. The result is a peripheral that can probably take any amount of mishandling, requiring a very determined schoolchild to damage it to any great extent.

On top of the casing are banks of minijack sockets that are divided into three groups. On the left of the interface are eight white input lines, with four earth lines, in green, positioned in between. Each jack is numbered 1, 2, 4, 8 and so on, corresponding to the number held in each of the bit positions of the register. Adjacent to the bit positions are LEDs that are used to inform you which of the bits are being held 'high' (indicating that a 5v current is being passed down that particular data line).



KEVIN JONES

Changes in the input lines can be registered by the computer, and the simplest way of demonstrating this is by setting all the lines to high. PEEKing the register will produce a value of 255, but if we connected a wire from one of the earth lines to one of the white input sockets, the voltage would drop to zero and the value of the register would drop accordingly, depending on which line was used. Thus, simple circuit breaking applications, such as burglar alarms, could be constructed.

THE OUTPUT LINES

To the right of the inputs, and covering most of the surface, are the output lines. Like the input lines, there are eight basic positions (numbered 1,2,4 and so on to 128), associated LEDs for each of the bit positions and four minijack sockets. Pairs of jack sockets are connected by means of relay circuits. POKEing a number into the register at address &FE61 will trigger the relays.

By connecting the electric motors and suitable power sources across the terminals, these relays can be used to drive the motors and position them as desired. Utilised in this way, up to four electric motors can be controlled from the Banana Interface at once. Obviously, the most effective way of controlling the electric motors (which may be connected to a floor turtle or robot arm) is by driving them through the output switches and monitoring their movements via the input ports.

Running along the top of the interface is another set of eight high-speed logic ports, each producing a current of 12v. These output lines are used to drive stepper motors from the interface, up to seven of which can be driven at any one time.

The Banana Interface would seem to have a market in schools and research establishments as an aid in teaching basic computer control of electrical devices. There are clearly a number of applications for which it could be used in demonstrations and experiments. It could also have wide applications for small businesses wishing to automate some of their production processes.

As far as the home hobbyist is concerned, the usefulness of such a device is a little less clear. It can certainly be used at home for a number of entertaining experiments and could probably teach you a lot about computer control techniques. As such, the serious robotics enthusiast might consider buying one. However, for BBC Micro and Commodore 64 owners with only a passing interest in the field of computer-controlled devices, the Banana Interface could well be too expensive for its limitations.

BANANA INTERFACE

PRICE
£205.85 (inc VAT)

DIMENSIONS
300×200×63mm

INTERFACES
40-way parallel connector from the printer and user ports of the BBC Micro and Commodore 64. Output ports, eight 12v output sockets

STRENGTHS
The Banana Interface is well made and allows for a wide variety of computer control applications

WEAKNESSES
The lack of a suitable 12v power source will limit its potential market either to committed hobbyists or educational and research establishments

DECLARING AN INTEREST

PROLOG solves problems by drawing from a 'database' of structures called 'facts' and applying them on a trial-and-error basis to the proposition in hand. We examine the declarative nature of the language, provide example programs and study how PROLOG 'backtracks' through a program when it reaches a proposition it cannot prove.

PROLOG is designed to give the programmer a means of describing the logical structure of a problem using a database of facts and 'if-then' rules. This 'declarative' statement of the problem is used by the deductive mechanisms within PROLOG to produce answers to the queries you have posed. The task of the programmer is thus transformed from one of telling the computer, in painstaking detail, how to use the operations of a language (to solve a predefined set of problems) to one of providing a clear and logical statement of the knowledge required to solve the problems, and then leaving PROLOG to carry out the task.

PROLOG has a very simple syntax but the terminology is rather obscure. The most important construct is the 'term'. A term can be a 'constant' (such as 'tree', 'john' or '25'), a 'variable' or a 'structure'. Structures are created from constants and variables. A very common structure in PROLOG programs is known as a 'fact'. This consists of a predicate (which must be a constant) either on its own, or followed by a list of arguments (which may be constants or variables) in brackets:

```
predicate1.
predicate2(argument1,argument2,
    argument3).
```

As we saw in the first instalment of our series, a predicate may be thought of as a relationship that exists between its arguments. In the English sentence 'Venusians eat trees', 'eat' is the predicate that relates Venusians to trees. In PROLOG, we could have written this fact as:

```
eat(venusians,trees).
```

Terms such as this can be accumulated to form large collections of facts in very much the same way as records are accumulated in database files. In the diagram, we have a database of facts about who eats what. Without any extra code, this database already gives us a PROLOG program.

To run the program, we need to set a proposition for the interpreter to prove.

Who Eats What?

```
eat(venusians,trees).
eat(jovians,rocks).
eat(earthlings,porridge).
eat(mercurians,worms).
eat(neptunians,porridge).
eat(martians,martians).
```

'Facts' in PROLOG are expressed as a predicate (in this case 'eat'), either on its own or followed by a list of arguments. They may be used to provide the program with a 'knowledge database' that can then be queried by the user

Let's say that we want to know whether Jovians eat rocks. We pose the query by typing:

```
?- eat(jovians,rocks).
```

after the system prompt (usually ?-). PROLOG then looks through our facts and, if it finds a match for the proposition, says yes, meaning 'yes, it can be shown from the program that eat(jovians,rocks) is true'. If we then ask:

```
?- eat(martians, porridge).
```

PROLOG will respond with no.

PROLOG VARIABLES

We can make our program do more for us by using variables in our query. The PROLOG convention is that variable names start with a capital letter while standard PROLOG constants start in lower-case. So, if we want to know who eats worms, we can ask:

```
?- eat(Creature,worms).
```

to which PROLOG will reply:

```
Creature = mercurians
```

and then pause, waiting for an input. This means that PROLOG has discovered that by setting the variable Creature to the value mercurians, it can prove that the proposition is true.

At this point, then, we can either enter RETURN, to say we are happy with the answer, or we can type a semicolon to instruct PROLOG to find another way of proving the proposition. In this case, however, there are no other ways, so PROLOG will respond with no. If we ask it to find who eats porridge, though, there will be two possible answers: earthlings and neptunians. So, if in response to:

```
?- eat(Creature,porridge).
```

PROLOG says:

```
Creature = earthlings
```

and we then type ;, PROLOG will respond with:

```
Creature = neptunians
```

If there were more solutions, pressing the semicolon after each would force PROLOG to find them all. How exactly this works we shall examine in the next instalment, but first let's look at three other important concepts in PROLOG — 'rules', 'inference chains' and 'backtracking'.

One or more terms make a clause, which may express a 'rule'. A clause always has a head consisting of one term and this may be followed by its body, having one or more terms. The head and the body are separated by the :— symbol operator, which is normally read as 'if'. Thus we get:

```
term1 :— term2,term3,term4.
```

The commas between terms in the body can be read as logical AND, so this example structure may be interpreted as signifying that 'term1 is valid IF term2 AND term3 AND term4 are all valid'.

To make this clearer, imagine we wanted to find out whether any of the creatures in our database were cannibals. We can write a clause that gives us a definition of a cannibal:

```
cannibal(Creature) :—
    eat(Creature,Creature).
```

This is a clause with one term in its body and we can read it as a 'rule' that states: a creature is a cannibal if that creature eats the same kind of creature.

If we add this rule to our program and then ask:

```
?- cannibal(X).
```

PROLOG matches up this proposition with the head of our new clause. Because this new clause is not automatically true, it must first be proved that the terms in the body of the clause are true. So PROLOG takes each one in turn, from left to right, and sets them up as propositions, just as though they had been typed in as queries. In this case there is only one term in the body:

```
..., eat(Creature,Creature).
```

and this matches the fact eat(martians,martians). This sets Creature = martians, which means that cannibal(martians) can be proved. This in turn sets X = martians (X was the variable name in the original query) and PROLOG will reply:

```
X = martians
```

In this case, eats(martians,martians) was found as a fact in the database. If, however, it had not been a simple fact but had been another rule, PROLOG would have had to set itself further sub-propositions to try to prove the truth of the head of the rule.

The PROLOG program in the diagram illustrates how this could happen. If we pose the query:

```
?- colour_of(marty,Colour).
```

wanting to find out the colour of marty, PROLOG first takes our query as its proposition. It then finds a rule to decide whether colour_of(marty,pink) is true and finds that this is so if (:—) the mood of marty is happy. Setting this as its next proposition then, it discovers a rule that says marty is happy if he can program in PROLOG. So can_program_in(marty,prolog) becomes the next proposition.

This could continue on down for a great many levels but, in this case, it stops here because there is no rule or fact that either proves, or shows how to prove, that marty is a PROLOG programmer.

PROLOG does not simply give up at this point. What it does, however, is admit its failure on the current proposition and then returns to the previous proposition to see if there is an alternative way of proving it. But that too fails (it cannot be shown that marty's mood is happy). So PROLOG *backtracks* again to see if there is an alternative way of proving the very first proposition. In fact, there is. PROLOG finds the fact colour_of(Martian,blue), sets Martian = marty and the proposition succeeds.

This process of working its way down through long chains of rules and setting itself a new proposition to prove at each step is how PROLOG solves all its queries. The method is called backtracking because if a particular path through the rules does not produce an answer, PROLOG will reverse up the path until it comes to an earlier choice-point, then carry on downward in this new direction. In this way it will either find its answer or explore every possible path through the rules. For this reason, PROLOG programs are best understood as 'trees', or chains of propositions, rather than as lists of statements.

In our next instalment of this series, we shall examine program flow and PROLOG's implementation of lists.

What Makes A Martian Blue?

```
colour_of(Martian,pink) :— mood_of
    (Martian,happy)
colour_of(Martian,blue)
mood_of(Martian,happy) :—
can_program_in(Martian,prolog).
mood_of(Martian,sad).

can_program_in(marty,basic).
can_program_in(miranda,prolog)
```

In this collection of terms, note that variable names commence with capital letters. The :— symbol may be interpreted as a logical 'if'

The Alvey Program

PROLOG currently plays a central role in the Alvey research program for the development of Fifth Generation 'supercomputers'. The program was established in 1983 following the report of the Alvey Committee, set up to advise on the scope for collaborative research in response to the initiative shown by the Japanese in their Fifth Generation Computer Project.

Collaboration is the keynote of the Alvey project — it has been closely linked with the European Fifth Generation project (ESPRIT) and moves have been made to establish links with the Japanese.

Approval has been given to approximately 100 projects, all of which are consistent with a central planning strategy that is reviewed and updated annually.

One aspect of the research involving PROLOG is the development of a PROLOG machine, which initial research suggests could be built using existing technology that would have a retail price substantially below that currently demanded for a typical minicomputer. Using PROLOG's 'intelligent database' characteristics, such a machine could be directly implemented as an expert system.

Other PROLOG applications currently being considered include its use as part of a natural language interface. PROLOG is particularly well suited to natural language processing, and since this area is an essential aspect of truly 'user-friendly' machines and fundamental to Fifth Generation research, PROLOG seems assured of a continued central role in the development of new technology

KAWALSKI: COURTESY OF COMPUTER WEEKLY

Parental Influence
Professor Bob Kowalski, often referred to as the 'father of logic programming', whose ideas on programming and logic systems were used by A. Colmerauer in the development of the first PROLOG interpreter at the University of Marseilles

PROFIT OR LOSS

We conclude our simulation of a 16th-century voyage to the New World by completing the trading and then sailing back home to sell the newly acquired goods. But you must first decide whether you want to involve yourself in a revolutionary uprising, from which you could make an enormous profit or lose everything!

As we enter the final phase of the game, trading has been completed and the ship has been loaded with the goods acquired from the chief of the New World inhabitants. On the following morning, the crew will restock with fresh water and food, and begin their voyage home. All the goods, except the guns, have been traded and the spices, carvings and pearls should sell for a profit when the ship returns to port.

Before the journey home, however, a further development ensues. During the night, a rival of the chief secretly visits the ship. The insurgent wants to buy your guns in order to overthrow the chief and offers 30 pearls for each one. There is a massive profit to be made if you accept the deal, but there are considerations to bear in mind.

If the chief discovers you have sold the guns, he will surely take revenge should the revolution be unsuccessful. However, the profit margin on the voyage could be so slim that you might consider the extra income worth the risk.

You can, of course, refuse the offer, in which case the chief will, as a gesture of appreciation, supply you with free provisions for the return journey and may even include 50 pearls as a reward. There is a chance of the revolution succeeding if you sell the guns, but if it fails, the consequences will be dire. The chief, after learning of your double-dealing, will order your ship to be set ablaze, thus ending the game.

If you can avoid this final disaster, the ship will set sail for home; the crew is in good shape, the winds are favourable and the journey will be completed in eight weeks. When the ship reaches port, the goods can be sold, the crew can be paid and, with luck, a profit will be made. Should there not be enough money to pay the crew, the ship can be sold to meet the costs. The final program report calculates a balance sheet for the voyage and displays information on whether you have made a profit or loss.

The main program, at line 893, calls a subroutine at line 10300, which deals with the offer made by the chief's rival to purchase your guns. The program first checks whether there are any guns available for trading, because if there are none, there will be no point in the revolutionary approaching the captain. Line 10305 looks at the second element of the supplies array, OA(2), which corresponds to the guns, and if this is equal to 0, there will, of course, be no guns and control will be returned to the main program. If there are guns available, the insurgent will make his offer.

You will be asked to type Y or N to the proposition at line 10326, and line 10330 will check to see if the input is correct. If not, it will loop back and prompt another response.

If you decide to sell the guns, the routine will jump to line 10400, where this can eventually be handled. Line 10405 generates a random number between 0 and 1, providing a 75 per cent chance of sending the program to line 10450, where the revolution will fail. If you feel the odds against the revolution succeeding are too high, they can be adjusted to suit your sense of fair play.

If the revolt does come to nothing, the chief will repossess the traded goods from the ship, set it alight and thus end the game. Otherwise, the program continues through to line 10429, in which case the revolution is successful, and the newly proclaimed chief will fully provision the ship before it sets sail.

ADJUSTING THE ARRAYS

The equation at line 10429 calculates the number of pearls obtained in the exchange by multiplying the number of guns, OA(2), by 30, and the subsequent total will be added to the first element of array AO() to record the amount of pearls purchased. Line 10430 removes the guns from the supplies array. For the purposes of the game, it is not strictly necessary to do this, but it is good programming practice and will make it easier to extend the game to include, say, incidents on the return voyage.

If you refuse to take advantage of the offer, the current chief may reward you with 50 pearls. This

Basic Flavours

Spectrum:
Replace AO() by E(), V1() by B() and V2() by D() throughout and make the following changes:

```
10310 CLS:GO SUB 9200
10328 INPUT I$:LET I$=I$( TO 1)
10354 LET I$=INKEY$:IF I$="" THEN GO TO
      10354
10400 CLS:GO SUB 9200
10501 CLS:GO SUB 9200
10547 LET I$=INKEY$:IF I$="" THEN GO TO
      10547
```

BBC Micro:
Make the following changes:

```
10310 CLS:GO SUB 9200
10354 I$=GET$
10400 CLS:GOSUB 9200
10501 CLS:GOSUB 9200
10547 I$=GET$
```

Module Thirteen: The Voyage Ends

Addition To Main Program Body

```
893 GOSUB10300
894 GOSUB10500
```

Revolution Routine

```
10300 REM REVOLUTION
10305 IFOA(2)=0THENRETURN
10310 PRINTCHR$(147):GOSUB9200
10315 S$="DURING THE NIGHT A RIVAL OF THE*":GOSUB9
100
10316 S$="CHIEF VISITS THE SHIP IN SECRET*":GOSUB9
100
10317 PRINT:GOSUB9200
10318 S$="HE WANTS TO BUY YOUR GUNS FOR*":GOSUB910
0
10320 S$="A REVOLUTION*":GOSUB9100
10322 PRINT:GOSUB9200
10324 S$="HE OFFERS YOU 30 PEARLS PER GUN*":GOSUB9
100
10326 S$="DO YOU SELL HIM THE GUNS?(Y/N)*":GOSUB91
00
10328 INPUTI$:I$=LEFT$(I$,1)
10330 IFI$<>"N"ANDI$<>"Y"THEN10328
10332 IFI$="Y"THEN10400
10334 PRINT:GOSUB9200
10336 S$="THE CHIEF FINDS OUT AND IS*":GOSUB9100
10338 S$="PLEASED WITH YOU*":GOSUB9100
10340 S$="HE GIVES YOU FREE PROVISIONS*":GOSUB9100
10342 S$="FOR THE JOURNEY BACK*":GOSUB9100
10344 GOSUB9200
10345 IF RND(1)<.75 THEN 10350
10346 S$="AND 50 PEARLS!!*":GOSUB9100
10348 AO(1)=AO(1)+50
10350 PRINT:GOSUB9200
10352 S$=K$:GOSUB9100
10354 GETI$:IFI$=""THEN10354
10359 RETURN
10400 PRINTCHR$(147):GOSUB9200
10405 IFRND(1)<.75THEN10450
10410 S$="THE REVOLUTION IS SUCCESSFUL*":GOSUB9100
10412 PRINT:GOSUB9200
10415 S$="THE NEW CHIEF REWARDS YOU WITH*":GOSUB91
00
10420 S$="FREE PROVISIONS FOR THE RETURN*":GOSUB91
00
10425 S$="JOURNEY.*":GOSUB9100
10429 AO(1)=AO(1)+(OA(2)*30):REM ADD PEARLS
10430 OA(2)=0
10431 GOTO10350
10450 S$="THE REVOLUTION FAILS!!*":GOSUB9100
10452 PRINT:GOSUB9200
10455 S$="THE OLD CHIEF IS ANGRY WITH YOU*":GOSUB9
100
10457 S$="HE BURNS YOUR SHIP AND STEALS*":GOSUB910
0
10458 S$="EVERYTHING!!*":GOSUB9100
10459 PRINT:GOSUB9200
10460 S$="   GAME OVER!!*":GOSUB9100
10462 END
10464 GOTO10462
```

End Of Voyage Routine

```
10500 REM END OF VOYAGE
10501 PRINTCHR$(147):GOSUB9200
10505 S$="WITH A STRONG CREW AND GOOD*":GOSUB9100
10507 S$="WINDS THE JOURNEY BACK GOES*":GOSUB9100
10508 S$="WELL AND TAKES ONLY 8 WEEKS*":GOSUB9100
10512 WW=0
10514 FORT=1TO5
10516 WW=WW+(8*CC(T)*WG(T))
10518 NEXT
10519 PRINT:GOSUB9200
10520 S$="WAGE BILL FOR RETURN VOYAGE = *":GOSUB91
00
10522 PRINTWW;"GOLD PIECES"
10524 PRINT:GOSUB9200
10526 S$="WHEN YOU GET BACK*":GOSUB9100
10528 S$="YOU SELL YOUR TRADING GOODS*":GOSUB9100
10530 S$="THE GOODS ARE NOW WORTH:*":GOSUB9100
10532 PRINT"PEARLS - ";V2(1);"GOLD PIECES"
10534 PRINT"CARVINGS - ";V2(2);"GOLD PIECES"
10536 PRINT"SPICES - ";V2(3);"GOLD PIECES"
10538 PRINT:S$="YOU GET A TOTAL OF*":GOSUB9100
10540 X=(AO(1)*V2(1))+(AO(2)*V2(2))+(AO(3)*V2(3))
10542 PRINTX;"GOLD PIECES"
10545 PRINT:S$=K$:GOSUB 9100:PRINT
10547 GET I$:IF I$="" THEN 10547
10550 S$="YOU NOW HAVE:*":GOSUB9100
10552 PRINTMO+X;"GOLD PIECES"
10555 PRINT:GOSUB9200
10556 S$="THE WAGE BILL FOR THE VOYAGE IS*":GOSUB9
100
10557 PRINTWT+WW;"GOLD PIECES"
10559 PRINT:GOSUB9200
10560 S$="YOU END THE GAME WITH:*":GOSUB9100
10562 Z=MO+X-WT-WW
10565 PRINTZ;"GOLD PIECES"
10566 PRINT:GOSUB9200
10567 PRINT:S$="YOUR RATING IS:*":GOSUB9100:PRINT
10568 IF Z>3200 THEN S$="ARCH CAPITALIST*":GOSUB 9
100:END
10569 IF Z>2500 THEN S$="MASTER TRADER*"GOSUB9100:
END
10570 IF Z>2000 THEN S$="MERCHANT CLASS III*":GOSU
B9100:END
10571 IF Z>1000 THEN S$="A PROPER JONAH*":GOSUB 91
00:END
10572 S$="MORE OF A DUCK THAN A DRAKE*"
10573 GOSUB9100:END
```

is randomly decided at line 10345. The final subroutine starts at line 10500 and deals with the return voyage and sale of goods.

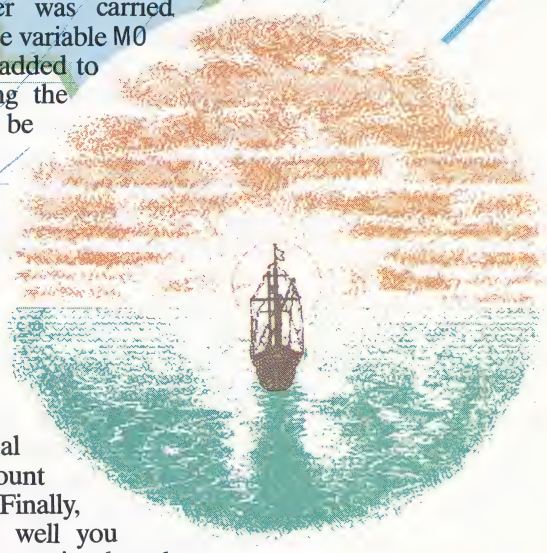
The return journey lasts eight weeks and the wage bill must be calculated. A variable, WW, is created at line 10512, which represents the wage bill for the return journey. Its value is calculated in the loop starting at line 10514. The formula multiplies the count of each crew type, CC(T), by the weekly wage rate for each, WG(T), to produce the weekly wage bill for all the crew of that type. The loop counter T ensures that this process is repeated for each crew type, eventually producing a total weekly wage bill for the entire crew. To calculate the bill for the return voyage, this new total will be multiplied by 8, and the program will print the total bill for the return voyage.

When the ship reaches port, the goods will be sold. Their market values are set by the array V2(3) and DIMensioned at line 62. These values were set randomly at the beginning of the program. Lines 10532 to 10536 of the end-of-voyage routine display the values held in this array, for each type of goods traded.

The total amount of gold received for the traded goods is calculated by line 10540 and stored in X. The formula used for this calculation multiplies the quantity of each commodity, recorded in the array AO(), by the current market value stored in V2(). The receipts from the pearls, carvings and spices are added and the total will be printed.

It is possible that not all the original capital of 2,000 gold pieces was spent at the start of the voyage. Any remainder was carried throughout the trip in the variable MO and the surplus will be added to the gold received during the trading; the amount will be displayed by line 10552.

The total wage bill for the voyage is calculated at line 10557 by simply adding the wage bill for the outward voyage, WT, to the bill for the return voyage. The program then calculates the trading profit by subtracting the total wage bill from the amount of gold in the coffers. Finally, the game assesses how well you have done by producing a rating based on the amount of profit you have made.



IAN McKINNELL

SPECTRUM ANALYSIS

In this introduction to a series of articles exploring aspects of the Sinclair Spectrum's operating system, we begin by inspecting the layout of the computer's memory map. We also include a discussion of the most favourable places to position machine code programs and look at some uses of several system variables.

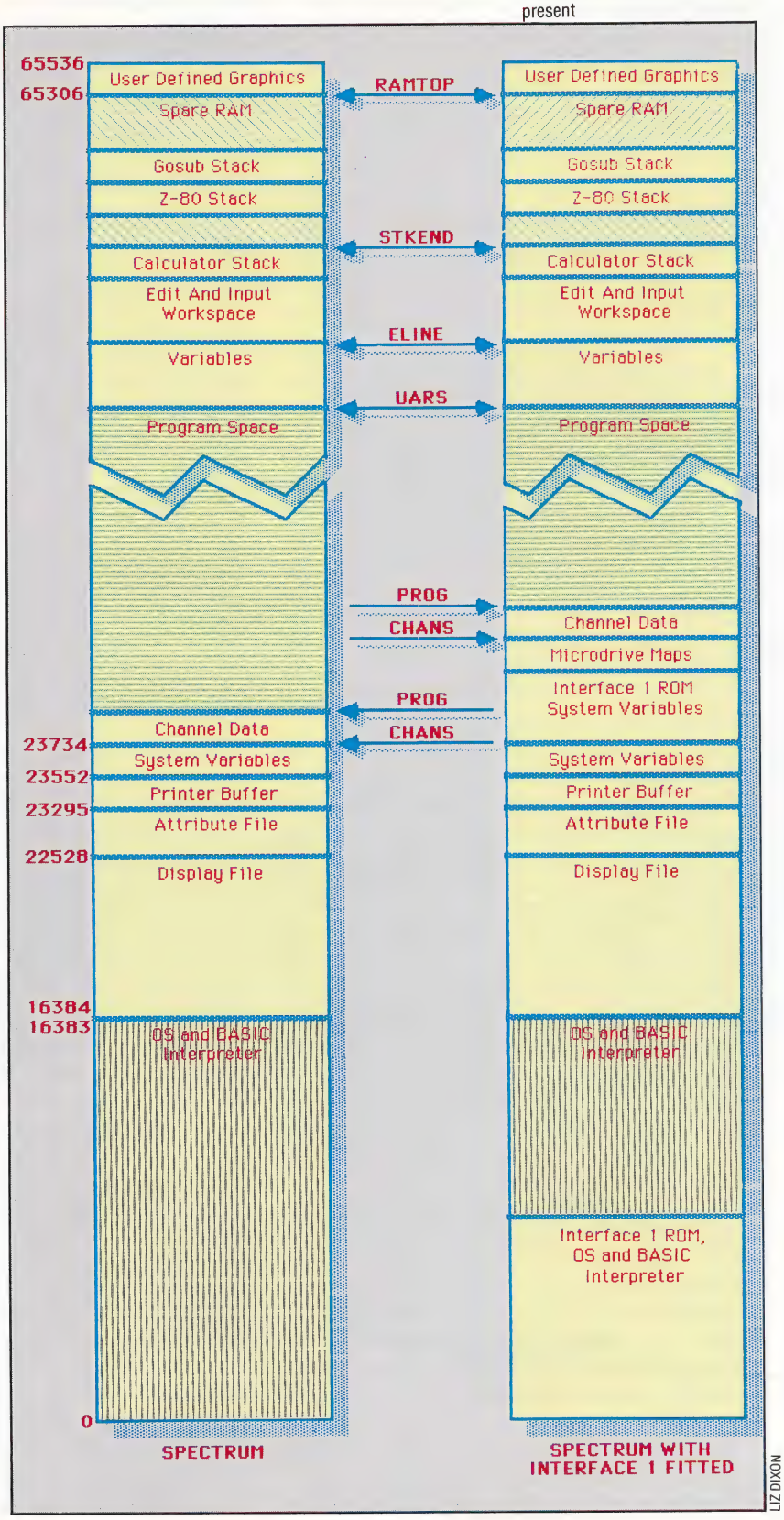
The Sinclair Spectrum, unlike the BBC Micro, has no separate operating system ROM inside the computer, since the OS and BASIC interpreter are both in the same chip. Furthermore, there has only even been one version of this ROM and since the Spectrum+ uses the same chip as the Spectrum this situation is unlikely to change. The OS is a compact set of routines — about half the size of the BBC OS — which takes up only about seven Kbytes of the Spectrum's 16Kbyte ROM chip.

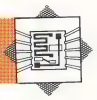
Despite the lack of variations in the operating system and the compact nature of the coding, the Spectrum OS suffers from one serious drawback in comparison to that of the BBC Micro. It is not vectored, apart from a couple of minor exceptions, and it is therefore rather more difficult to alter the way in which the OS handles certain events. It is, however, possible to get round this and we shall be tackling this problem later in the course.

If you have followed the series on the BBC Micro's operating system (beginning on page 858), you will certainly be familiar with the role that the operating system plays within the computer. Briefly, this role can be summed up as the handling of all routines concerned with input, output, display and filing, interrupt-handling, and generally providing an interface between the hardware and the user program or BASIC interpreter. We'll begin our examination of the Spectrum OS by considering the computer's memory map, which is shown.

The first 16 Kbyte block of memory is occupied by a ROM chip holding the BASIC interpreter and operating system. Roughly speaking, the OS occupies the lower six or seven Kbytes of the ROM and the BASIC interpreter uses the upper part of the ROM chip. When an Interface 1 is being used, the lower eight Kbytes of memory are paged in a similar way to the paging of the area of memory between &8000 and &BFFF on the BBC Micro. When you want to access any of the facilities offered by the Interface 1 — such as Microdrives, serial interface or Local Area Network — the normal ROM is paged out and the Interface 1 ROM is paged in. We'll look at this in more detail in a future instalment of the course.

Inside Data
Our memory map for the 48K Spectrum (and the Spectrum+) shows how fitting an Interface 1 reconfigures some areas of the Spectrum RAM. The system variables area normally extends to around 23733 but will be extended if Interface 1 is present





The rest of the memory space is taken up by RAM, and as you can see there are slight differences in the RAM utilisation on a Spectrum when the Interface 1 is being used.

Looking at sections of the map in turn, the *display file* holds information regarding the shape of the images that are to be displayed on the screen. The data referring to colour, BRIGHT and FLASH is not held in this file, but is stored in the *attribute file* instead, because this information refers to a character square on the display screen. The *printer buffer* is a 256-byte area of memory used by the ZX printer when hard copy is requested.

The *system variables* are used by the OS and interpreter so that they can oversee what is happening in the system. We'll look at some of the more useful of these system variables throughout this series. A full list of them can be found in the Spectrum's user manual on pages 127 to 130. Unfortunately, no such list is given in the Spectrum+ manual.

This is where the differences between a Spectrum with and without the Interface 1 are most obvious. Not surprisingly, the Interface 1 ROM requires some workspace and system variables for its work, and so when it is used it generates extra system variables — and these are located after the normal Spectrum system variables. Clearly, this exerts a 'knock on' effect on the rest of the memory in use, and adjustments are made accordingly.

The *Microdrive maps* are again specific to Interface 1 and are used so that the Interface can keep track of used and unused areas of a Microdrive tape.

The *channel data* area provides us with a limited form of vectoring for the LLIST, LPRINT, INPUT# and PRINT# commands. For example, we can write a machine code program to drive a standard printer and alter the channel data so that when an LPRINT or LLIST instruction is encountered it is acted upon by our piece of machine code, rather than the usual BASIC interpreter routines. We'll take a detailed look at this in the next instalment of the course. The channel data is also used by Interface 1.

We then get to the BASIC program and variable workspace, used by the system for storing the text of a BASIC program and the variables used by that program.

The *edit* and *input* workspaces are used by the system to EDIT a program line, or to input data or program lines. Once editing or text entry has finished, the program lines are stored in the appropriate place in the program, and these areas are free to be used again.

We then encounter a group of stacks. The *calculator stack* is used by the BASIC interpreter for its arithmetical operations. The *Z80 stack* is used by the CPU in much the same way as the 6502 uses page one of its memory — that is, to store return addresses for machine code subroutines and data that has been pushed onto the stack for temporary storage. The *GOSUB stack* is again part of the memory used by the BASIC interpreter. It stores return addresses for BASIC GOSUB statements. Finally, the area for *user-defined graphics* holds the data, obviously, for characters defined by the user.

WHERE TO PUT MACHINE CODE

One thing that becomes clear from the memory map is that no memory is reserved for user machine code programs. This is also true, of course, for the BBC Micro after a disk drive or Econet interface is fitted. On the Spectrum there are three possible areas to put your code. These are in a line 1 REM statement — we can work out the position of line 1 in memory from the value held in the system variable PROG. Adding five to this value gives you the address of the first character in a line 1 REM statement.

The other locations are the printer buffer and between the RAMTOP and the area of memory used for user-defined graphics data storage. This latter method is similar to the use of memory between HIMEM and the start of video RAM in the BBC Micro. The following table shows the relative merits of these methods:

Place	Spectrum	Spectrum & Interface 1
Line 1 REM	✓	✗
Printer Buffer	(✓)*	(✓)*
Above RAMTOP	✓	✓

*This should NOT be used if a printer is in use

As you can see, only the last option is worth considering as the other two methods can be risky in certain circumstances. Anything above RAMTOP is safe from possible overwriting by BASIC, and reserving space is rather easy. The exact position of RAMTOP is held as a two-byte variable at addresses 23730 and 23731. The value is stored in standard Z80 format — lo-byte followed by hi-byte — so to check the current position of RAMTOP you should execute:

```
LET ramtop=PEEK23730+256*PEEK23731
```

Once you have the position of RAMTOP noted down (it's usually 65306 on a standard 'clean' 48 Kbyte machine), you should estimate how many bytes you will need for your machine code program. Having worked out the value using the formula RAMTOP-(number-of-bytes-in-program + 1), you can then lower RAMTOP by using the CLEAR command:

```
CLEAR value
```

Note that if you use this command within a BASIC program, it will perform a CLS, reset the PLOT position, and carry out a RESTORE. It will also clear and relocate the GOSUB stack. For this reason, you should make a habit of ensuring that the command is used carefully and located whenever possible at the beginning of a program — otherwise problems will almost certainly ensue.

Having executed the CLEAR value command, the

new position of RAMTOP will now be stored at locations 23730 and 23731, and the first byte available for your machine code program will be at address (value+1). Thus, assuming a clean 48 Kbyte machine, the command CLEAR 59999 will reserve about five Kbytes of memory for your programs, starting with location 60000. Once memory has been reserved in this manner, the POKE command can be used to store the bytes that make up your program in memory.

SOME SYSTEM VARIABLES

We'll finish this introduction to the Spectrum OS by looking at some of the system variables that hold information about the way in which memory is allocated in the system. Accessing System Variables has to be performed by PEEKing or POKEing the appropriate memory locations — there is no equivalent of the OSBYTE or OSWORD calls used by the BBC Micro.

- chars: This variable, at addresses 23606 and 23607, points to a memory location 256 bytes below the character set information. Normally, the address held in this two-byte variable points to an address in ROM 256 bytes below the address of the first byte of the definition of the space character (CHR$32). The actual address held in this system variable, like the others, can be evaluated by:

```
LET chars=PEEK 23606+256*PEEK 23607
```

If you want to, you can alter the value held in this variable to point to information defining a new character set, which can be stored in RAM.

- vars: This is found at locations 23627 and 23628, and holds the start address of the BASIC variables in memory. You can PEEK out the values held here, but *do not* alter these values, or else you could cause the Spectrum to lose track of its variables! The line:

```
LET vars=PEEK 23627 + 256*PEEK 23628
```

gives you the start address of variables.

- prog: This is a system variable that holds the start address of the BASIC program. It is, therefore, similar to the BBC Micro's system variable PAGE. It is stored in locations 23635 and 23636, so we can work out the start of program address by typing:

```
LET prog=PEEK 23635+256*PEEK 23636
```

If you evaluate vars, as shown above, then:

```
PRINT vars—prog
```

will return the length of your program.

- eline: This is stored at addresses 23641 and 23642, and holds the address of the start of any text being input to the system. We can use it to work out the amount of memory taken up by the variables in use. Assuming you've evaluated vars already:

```
LET eline=PEEK 23641+256*PEEK 23642
PRINT eline—vars
```

will return the amount of memory taken up by your variables.

- chans: A variable that holds the address of the start of channel information. It is at addresses 23631 and 23632. We'll look at the subject of channels in greater detail in a future instalment in the series.
- stkend: This is held at locations 23653 and 23654. The variable, in conjunction with the value of RAMTOP, enables us to estimate the amount of memory left for our BASIC program and variables. This value is evaluated with:

```
LET stkend+PEEK 23653+256*PEEK 23654
```

If we've already evaluated stkend, then the amount of memory remaining is given by:

```
PRINT ramtop—stkend
```

Alternatively, we can use a ROM routine to give us an estimate of the amount of memory left. This method, however, is not as accurate as the one we've just used. Assuming we've got the current value of RAMTOP:

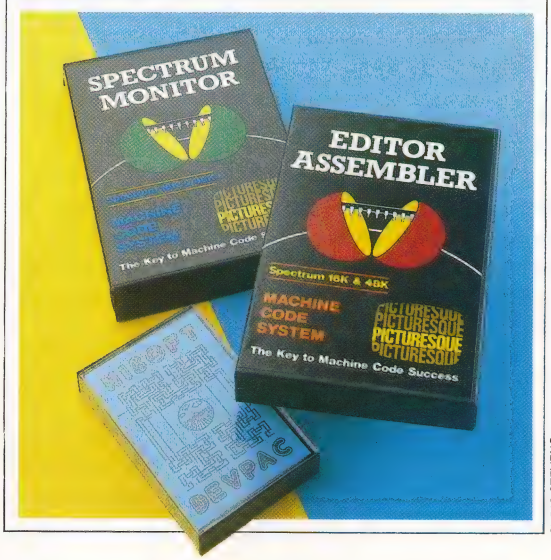
```
PRINT ramtop—USR 7962
```

will give an estimate of free space.

Assembly Time

There are a number of assembler packages available for the Sinclair Spectrum user interested in delving into machine code. One of the most popular of these is the DEVPAC 3 from Hisoft. Provided in cassette form, it includes the GENS3 assembler and MON3 monitor/disassembler. A Microdrive version of the package is also available. Although widely used by Spectrum owners, DEVPAC 3 is not one of the friendliest assemblers on the market; however, at £14, it is one of the least expensive.

The Picturesque Editor Assembler and Spectrum Monitor (including a disassembler) is a double package, priced at £8.50 and £7.50 respectively. One of the main advantages of the Picturesque system is that it can be fitted comfortably onto a 16 Kbyte Spectrum. The manuals that come with this package are superior to the documentation provided by Hisoft, and the whole system has a much more professional feel



CHRIS STEVENS

Measuring Stick

One of the features lacking in the Spectrum is a convenient means of calculating the length of a program; it can be done using PEEK, but here is a small utility that uses the system variables PROG and VARS to calculate the length of whatever BASIC program is currently in memory, and store it in a line 1 REM statement. All that is required is that the first line of the program is a REM statement containing at least five characters, such as:

```
1 REM 00000 Bytes Long
```

When the machine code routine is executed, the REM statement will be modified to contain the length of the program. For example:

```
1 REM 00801 Bytes Long
```

This is rather useful, since the next time the program is examined, the length will be present. The first piece of code is an assembly listing of the machine code involved. If you do not have a suitable assembler, the second listing is a BASIC program that can be used to load the machine code program.

In the BASIC Loader, CLEAR 61999 sets the value of the System Variable RAMTOP to 61999. The first 'safe' byte is therefore at address 62000, and we have used this as the start address for the machine code routine. RAMTOP will remain at this new value until it is altered with a CLEAR n command, a PRINT USR 0 command is executed or the machine is turned off and then on again. NEW will not affect the value of RAMTOP, and the machine code routine is protected from being overwritten by BASIC while RAMTOP remains unaltered.

Once the program is in memory, USR is used to run the machine code. The argument of the USR(n) function is the address of the machine language routine that you want to execute. This address can be anywhere in memory. Thus, all the following will run the machine code at address 62000, and any of them could be used to execute the machine code program we have just looked at:

```
RANDOMISE USR(62000)
PRINT USER(62000)
LET L=USR(62000)
```

The PRINT and LET methods of using USR also return a result to BASIC that can be made use of. The result is the value that is in the BC register pair when the machine code routine finishes and returns to BASIC; a machine code routine can thus pass results back to BASIC. If your routine does not affect the value of the BC register, the address of the routine is returned.

Assembly Listing

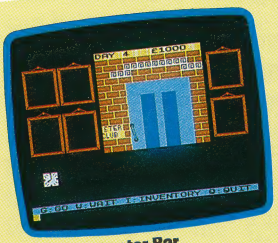
```
62000                 10          ORG 62000
62000  2A4B5C         20          LD    HL,(23627)  ;get VARS in HL
62003  ED5B535C       30          LD    DE,(23635)  ;get PROG in DE
62007  AF             40          XOR   A           ;clear carry flag
62008  ED52           50          SBC   HL,DE       ;get program length into HL
62010  DD2A535C       60          LD    IX,(23635)  ;start of program into IX
62014  DD23           70          INC   IX
62016  DD23           80          INC   IX          ;increment IX to point to
62018  DD23           90          INC   IX          ;character after REM token
62020  DD23          100          INC   IX          ;in first program line
62022  DD23          110          INC   IX
62024  011027        120          LD    BC,10000    ;start getting ASCII codes
62027  CD6BF2        130          CALL  PRINT       ;of each digit in program
62030  DD23          140          INC   IX          ;length and store in REM
62032  01E803        150          LD    BC,1000     ;statement
62035  CD6BF2        160          CALL  PRINT
62038  DD23          170          INC   IX
62040  016400        180          LD    BC,100
62043  CD6BF2        190          CALL  PRINT
62046  DD23          200          INC   IX
62048  010A00        210          LD    BC,10
62051  CD6BF2        220          CALL  PRINT
62054  DD23          230          INC   IX
62056  010100        240          LD    BC,1
62059  AF            250  PRINT:   XOR   A           ;clear carry and xero counter
62060  B7            260  L1:     OR    A
62061  ED42          270          SBC   HL,BC       ;subtract power of ten from
                                                     ;program length...
62063  3803          280          JR    C,FINISH    ;and jump if negative
62065  3C            290          INC   A           ;inc A reg if not negative
62066  18F8          300          JR    L1          ;repeat
62068  09            310  FINISH: ADD   HL,BC       ;restore prog length to
                                                     ;positive value for the next
                                                     ;power of ten
62069  C630          320          ADD   A,48        ;get ASCII code of digit
62071  DD7700        330          LD    (IX),A      ;store ASCII code in REM
62074  C9            340          RET               ;finished
```

BASIC Loader

```
  10 REM 00000 Bytes Long
  20 CLEAR 61999
  30 FOR i=0 TO 74
  40 READ a: POKE 62000+i,a
  50 NEXT i
1000 DATA 42,75,92,237,91,83,92,175,237,82,221,42,
83,92,221,35,221,35,221,35,221,35,1,16,39,2
05,107,242,221,35,1,232,3,205,107,242,221,35,1,100
,0
1010 DATA 205,107,242,221,35,1,10,0,205,107,242,22
1,35,1,1,00,175,183,237,66,56,3,60,24,-8,9,198,48,
221,119,0,201
```

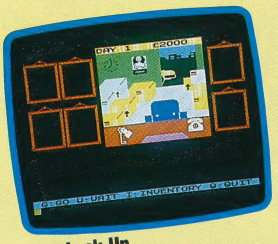
Note: To ensure that Measuring Stick locates the correct position for inserting the digits indicating program length, do not insert any spaces between the first line number and the REM keyword (other than the space automatically inserted by the Spectrum keyword system)

NICE LITTLE EARNER

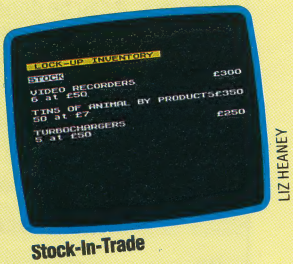


The Winchester Bar

Making A Deal



The Lock-Up



Stock-In-Trade

LIZ HEANEY

Making A Killing
Shown here are four scenes from the game Minder. The object of the game is simply to sell your stock (kept in your 'lock-up') to dealers. If you run out of stock, further goods can be bought from the customers of the Winchester Bar. Of course, you must bargain for the best possible price when both buying and selling in order to make the maximum amount of money in the time available

Over the past few years Arthur Daley and Terry McCann have become very popular characters in the television programme Minder. You can now take part in their exploits with a new game produced by Euston Films, in which you play the part of cigar-smoking Arthur, ever on the lookout for a shady deal and a quick profit.

The television programme Minder has been one of the greatest successes on British television in recent years and its conversion to game format is perhaps long overdue (the series began in the mid-1970s.)

Minder is based on the shady goings-on of Arthur Daley, a well-known dealer in dubious goods of every description, and Terry McCann, his 'minder' and henchman. Your role in this game is the part of Arthur, and the object is to make as much money as possible within a fortnight by buying and selling goods.

You must approach other dealers to whom you can sell your merchandise. You will often have some rather peculiar things to sell, such as tins of animal by-products or 'nucleonic ferkinators'. Just as often, the dealers will offer you too low a price for the goods and you will have to haggle with them to make it more reasonable.

One of the bigger thorns in Arthur's side is Detective Sergeant Chisholm, the local CID officer. DS Chisholm's influence begins when the dealer, whose premises you have entered in order to sell some home computers, which you *happen* to have acquired, informs you that the Detective Sergeant is in fact on the lookout for stolen computers. This news, of course, will immediately drop the price.

Arthur's problems are further compounded if DS Chisholm arrives at the 'lock-up' — the warehouse in which Arthur keeps his goods — and finds the stolen articles.

In order to make deals and find Terry, who you need to transport your goods to and from the lock-up, you will usually have to go to the Winchester, a drinking club. In this scene, as in any of the others in the game, there are up to six characters with whom you can talk. Their faces will appear in one of the picture frames around the screen, and in order to approach one of them, you merely press the key corresponding to that character.

Approaching characters in the Winchester is not really necessary, unless you want to talk to someone in particular, like Dave the barman or Terry — most of the others will try to sell you such worthwhile merchandise as mahogany rabbit hutches and polythene wet suits.

The strategy here is much the same as when trying to sell. You first find out how much your contact is selling these goods for and how many he has, and then you can begin to haggle. As you negotiate a price, the clock above you ticks away — the contact will leave if the deal is not made within a certain time. If you decide that you don't want to haggle any more, you can type in BYE and the person will go away.

Terry is something of an elusive character and you may not be able to contact him for some time, and when he completes a job he will expect a favour in return — either a drink or a bonus. Thus when you are negotiating a deal you have to bear in mind that Terry must be paid out of the profits.

Without doubt, the most amusing part of this game is the negotiation. While you and your opponent haggle over the price, the screen will display a string of ploys like 'lovely little earner' or 'fantastic quality'. When you offer a price, it is important that you type in the correct syntax, such as 'I'm offering 20 quid'. Otherwise, the computer may not understand you.

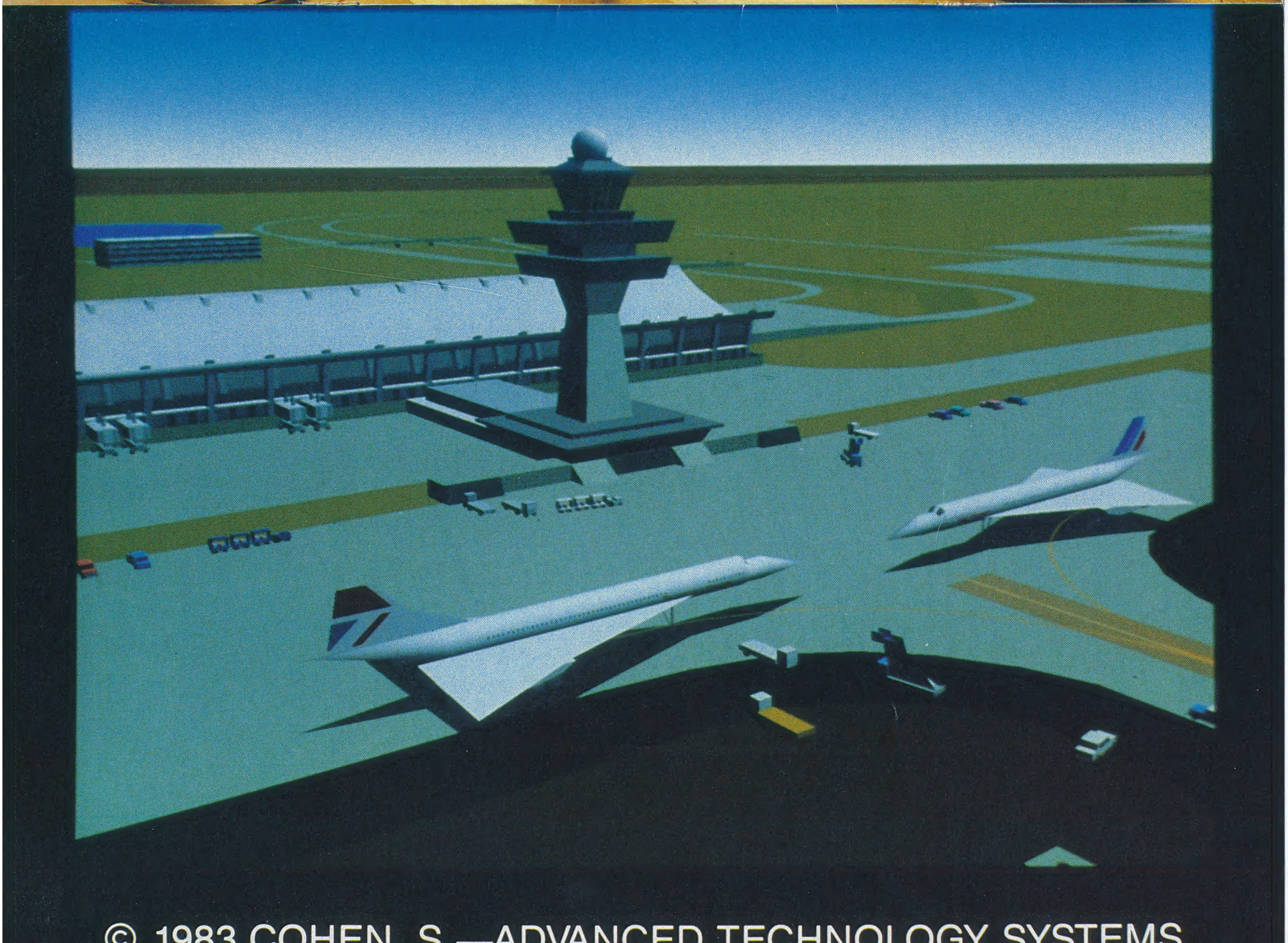
Unlike many games of this nature, Minder is as amusing and interesting as the television series. It manages to capture the flavour of the programme accurately and should prove to be as popular with the games enthusiasts as with those who enjoy the television series.

Minder: For the Sinclair Spectrum, MSX, Memotech, Amstrad and Commodore 64
Price: £9.95
Publishers: DK'tronics, Shire Hill Industrial Estate, Saffron Walden, Essex CB11 3AQ
Author: Don Priestley
Format: Cassette
Joysticks: Not required

DATABASE

Here, courtesy of Commodore Business Machines, we publish another instalment of the Commodore 64's memory map

LABEL	HEX ADDRESS	DECIMAL LOCATION	DESCRIPTION
DFLTN	0099	153	Default Input Device (0)
DFLTO	009A	154	Default Output (CMD) Device (3)
PRTY	009B	155	Tape Character Parity
DPSW	009C	156	Flag: Tape Byte-Received
MSGFLG	009D	157	Flag: $80 = Direct Mode, $00 = Program
PTR1	009E	158	Tape Pass 1 Error Log
PTR2	009F	159	Tape Pass 2 Error Log
TIME	00A0–00A2	160–162	Real-Time Jiffy Clock (approx) 1/60 Sec
	00A3–00A4	163–164	Temp Data Area
CNTDN	00A5	165	Cassette Sync Countdown
BUFPNT	00A6	166	Pointer: Tape I/O Buffer
INBIT	00A7	167	RS-232 Input Bits / Cassette Temp
BITCI	00A8	168	RS-232 Input Bit Count / Cassette Temp
RINONE	00A9	169	RS-232 Flag: Check for Start Bit
RIDATA	00AA	170	RS-232 Input Byte Buffer/Cassette Temp
RIPRTY	00AB	171	RS-232 Input Parity / Cassette Short Cnt
SAL	00AC–00AD	172–173	Pointer: Tape Buffer/ Screen Scrolling
EAL	00AE–00AF	174–175	Tape End Addresses/End of Program
CMP0	00B0–00B1	176–177	Tape Timing Constants
TAPE1	00B2–00B3	178–179	Pointer: Start of Tape Buffer
BITTS	00B4	180	RS-232 Out Bit Count / Cassette Temp
NXTBIT	00B5	181	RS-232 Next Bit to Send/ Tape EOT Flag
RODATA	00B6	182	RS-232 Out Byte Buffer
FNLEN	00B7	183	Length of Current File Name
LA	00B8	184	Current Logical File Number



© 1983 COHEN, S.—ADVANCED TECHNOLOGY SYSTEMS